Examination of Hypercube Implementations of Genetic Algorithms

THESIS

Andrew Dymek
Captain, USAF

AFIT/GCS/ENG/92M-02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/92M-02

An Examination of Hypercube Implementations of Genetic Algorithms

THESIS

Andrew Dymek
Captain, USAF

AFIT/GCS/ENG/92M-02

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1992	Master's Thesis

4. TITLE AND SUBTITLE

An Examination of Hypercube Implementations of Genetic Algorithms

5. FUNDING NUMBERS

6. AUTHOR(S)

Andrew Dymek, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/92M-02

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Genetic algorithms are stochastic search algorithms which model natural adaptive systems. In support of the development of a genetic search package for AFIT's iPSC/2 Hypercube, this study focused on two problem areas associated with hypercube implementations.

Premature convergence occurs when the "population" becomes dominated by locally optimal, but globally inferior, solutions. Based on an examination of past hypercube implementations, the selection and communication strategies were hypothesized as causes of premature convergence. Experiments to test these hypotheses were conducted on Rosenbrock's saddle, a function often associated with premature convergence. Communication of best solutions led to premature convergence in small population sizes, but increased the likelihood of finding the global optimal in large population sizes. Genetic algorithms using global selection were more robust than those using local selection.

GA-hard problems are intrinsically difficult for standard genetic algorithms. Messy genetic algorithms are effective against GA-hard problems. The second part of this study added a parallel version of a messy genetic algorithm to the genetic algorithm package. Against a sample GA-hard problem, the parallel implementation achieved a linear speedup of the sequential bottleneck while still finding the global optimal. The messy genetic algorithm should be applied to problems of practical importance.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Genetic Algorithms, Hypercube	189
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

AFIT/GCS/ENG/92M-02

An Examination of Hypercube Implementations of Genetic Algorithms

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Andrew Dymek, B.S.

Captain, USAF

March, 1992

Preface

Genetic algorithms are stochastic search algorithms which model the natural selection process. The purpose of this study was to continue development of a genetic algorithm package for use on AFIT's iPSC/2 Hypercube computer. As the goal of the package is to provide a robust search package, the research focused on two problems areas not adequately addressed by the current package.

The first problem area examined is known as premature convergence, the tendency of a genetic algorithm to return locally optimal solutions. A goal of this study was to provide guidelines on how to reduce premature convergence. Rosenbrock's saddle, a function against which sequential and parallel genetic algorithms often prematurely converge, was selected as the target problem. The parallel decomposition used in past implementations was examined in detail. The communication of solutions, use of local selection, and small population sizes were thought to be likely causes of premature convergence. Experiments were conducted which included the past implementation as well as implementations using global selection and no communication of solutions. The population size was varied over a wide range. The genetic algorithms which communicated the best solution had a greater likelihood of finding the global optimal at the upper ranges of the populations sizes. However, at smaller population sizes, sharing of the best solutions seemed to increase premature convergence. The strategies using global selection seemed more robust than the strategies using local selection over the range of population sizes used in this study. While it is impossible to guarantee the strategies found to reduce premature convergence against Rosenbrock's saddle would reduce premature convergence in any problem, there is some hope in their general applicability given that Rosenbrock's saddle is a very difficult problem for a genetic algorithm.

However, the strategies would not likely work against GA-hard (genetic algorithm-hard) problems. GA-hard problems are intrinsically difficult for standard genetic algorithms. To deal effectively with GA-hard problems, a new genetic algorithm, called a messy genetic algorithm, has

been developed by Goldberg, Korb, and Deb. The second part of the study focused on adding a parallel version of a messy genetic algorithm to the genetic algorithm package. A parallel version is attractive given the large memory resources and sequential bottleneck associated with a messy genetic algorithm. Against a sample GA-hard problem, the parallel implementation achieved a linear speedup of the sequential bottleneck while still finding the global optimal. This work should be continued against problems of practical importance, especially given the bold conjectures by Goldberg, Korb, and Deb regarding the effectiveness of messy genetic algorithms against combinatoric optimization problems.

In the course of this study, I have received a good deal of help and guidance. My thesis advisor, Dr. Gary Lamont, suggested the topic and offered support throughout. Maj William Hobart and Maj Paul Bailor provided valuable comments as readers. Mr. Richard Norris, the iPSC/2 system manager, was always willing to answer questions on the C programming language. Capt Paul Hardy and Capt Joann Sartor taught me much of what I know of LaTeX and its related facilities. To all these people I give my sincere thanks.

Andrew Dymek

Table of Contents

List of Figures

Abstract

Genetic algorithms are stochastic search algorithms which model natural adaptive systems. In support of the development of a genetic search package for AFIT's iPSC/2 Hypercube, this study focused on two problem areas associated with hypercube implementations.

Premature convergence occurs when the "population" becomes dominated by locally optimal, but globally inferior, solutions. Based on an examination of past hypercube implementations, the selection and communication strategies were hypothesized as causes of premature convergence. Experiments to test these hypotheses were conducted on Rosenbrock's saddle, a function often associated with premature convergence. Communication of best solutions led to premature convergence in small population sizes, but increased the likelihood of finding the global optimal in large population sizes. Genetic algorithms using global selection were more robust than those using local selection.

GA-hard problems are intrinsically difficult for standard genetic algorithms. Messy genetic algorithms are effective against GA-hard problems. The second part of this study added a parallel version of a messy genetic algorithm to the genetic algorithm package. Against a sample GA-hard problem, the parallel implementation achieved a linear speedup of the sequential bottleneck while still finding the global optimal. The messy genetic algorithm should be applied to problems of practical importance.

An Examination of Hypercube Implementations of Genetic Algorithms

I. Introduction

1.1 General Issue.

Computer solutions to many problems cannot be obtained in acceptable amounts of time. Even the most powerful current computer, if given a problem sufficiently complex, would take centuries to return a solution. One might think a TeraFLOP computer[1], the ultimate goal of DARPA's[2] Touchstone Program, would provide the final answer to this computational problem. Yet if the truly complex problems—the "Grand Challenges" (62)—are ever to be solved, improved algorithmic methods must accompany improvements in computer technology (75:179).

Even with a process as seemingly mundane as a search, the state of the art in algorithms cannot be said to be acceptable in many cases. For example, a typical solution method (algorithm) for combinatorial optimization problems can involve an exhaustive search of all possible solutions. While this guarantees an optimal solution, search times grow exponentially with the size of the problem. Even a TeraFLOP computer would quickly succumb to moderately-sized problems having an exponential time complexity. The execution times shown in Figure 1 are especially disconcerting in light of the fact that a search is often involved in the solution of many critical problems.

To allow a search to complete in a reasonable amount of time, often the requirement to find the optimal solution is relaxed. Search algorithms that sacrifice solution quality for efficiency are often called semi-optimal search algorithms. Semi-optimal search methods achieve greater efficiency (generally polynomial-time solutions) by searching only a subset of the possible solutions. The risk

[1] A computer capable of one trillion floating point operations per second
[2] Defense Advanced Research Projects Agency

Time Complexity Function	*Problem Size (n)*				
	30	40	50	60	70
2^n	$1.07 * 10^{-3}$ sec	1.09 sec	18.7 min	13.35 hrs	36.35 yrs
3^n	3.41 min	140.8 days	200 centuries	$1.3 * 10^7$ centuries	$1.34 * 10^{13}$ centuries

Figure 1. Solution times for exponential time complexity algorithms on a TeraFLOP computer. Adapted from (28:11).

is that the solution returned by a semi-optimal search method may be quite inferior to the global optimal solution.

An additional limitation of most semi-optimal search methods is lack of robustness (34:2-7). One common semi-optimal method known as a greedy algorithm may return an outstanding solution or a miserable solution depending on the distribution of solutions in the search space. Gradient-based search techniques perform well against differentiable functions, but have difficulties with functions having no analytical derivative. A search algorithm that performs efficiently over all problem domains has yet to be found. However, one fairly new class of algorithms called genetic algorithms seems to come close.

Genetic algorithms, adaptive search algorithms which model the natural selection process, have shown much promise in terms of efficiency, effectiveness, and robustness. These algorithms have returned excellent solutions to problems in such diverse fields as combinatoric and functional optimization, aircraft design, and conformational analysis, just to name a few. In addition to empirical evidence demonstrating robustness, there is a theoretical foundation as well (47) (34:2). Also, the polynomial time complexity of genetic algorithms makes them less sensitive to increases in problem dimensionality than a search algorithm having exponential time complexity.

Despite the generally outstanding performance of genetic algorithms, problems and limitations remain. For instance, while genetic algorithms have been quite successful at returning near-optimal solutions in diverse applications, an occasional tendency to return inferior solutions has been observed. A common reason cited for such problems is *premature convergence*, defined as

dominance of the solution "population" by local optima. Premature convergence has been observed in both sequential and parallel implementations. Certain problems, known as GA-hard (Genetic Algorithm-hard) problems are inherently difficult for the standard genetic algorithm.

To explore the capabilities of genetic algorithms, a Genetic Algorithm Toolbox is being developed here at AFIT for use on the iPSC/2 and iPSC/860 Hypercubes. To date, the standard genetic algorithm has been parallelized (71). The goal is a robust search package. This thesis research explores means of attacking premature convergence and GA-hard problems.

1.2 Background

There is a vast array of search methods (algorithms). Despite the apparent diversity in their operation, many search methods can be described in terms of of the following components (7:79-80):

- A set of candidates from which to construct a solution.
- A set of candidates that have already been used or considered.
- A test for feasibility.
- A selection function used to choose a the next candidate in the composition of the solution.
- An objective function which assigns a value to a solution.

Search algorithms based on such components tend to proceed in an orderly, deterministic manner. Bearing witness to the orderliness of the search process is the fact that it is often described with a tree or a graph. Such popular search algorithms as the A*, AO*, best-first, and branch-and-bound, are included in this class of orderly, deterministic algorithms.

A genetic algorithm differs quite significantly from the search process described above. Only the set of candidates and a objective function can be found in a genetic algorithm. These components take on the syntax and some of the semantics of analogous components in the natural selection process. The set of candidates are encoded into a string of genes amenable to genetic

operations. The objective (fitness) function provides a measure of the fitness of a solution. Here the similarities end. A main contributor to the differences are the genetic operators. The stochastic and disruptive nature of genetic operators leads to a much less structured search than, say, a greedy or best-first search. As a consequence, a genetic algorithm has neither a test for feasibility, a selection function, or a set of used candidates, nor can the course of the search be described as a tree or a graph. Additionally, a genetic search considers multiple solutions (the population) at the same time, another key difference from most search algorithms. Yet while a genetic algorithm might seem a radical departure from traditional methods, the differences apparently do not prevent genetic algorithms from being an effective search technique.

Genetic algorithms have been used as a search strategy in diverse problem domains (Figure 2) since their introduction in 1970's. John Grefenstette of the Navy Labs developed a C implementation of a genetic algorithm in the mid 1980's (41). Designed to facilitate experimentation, Grefenstette's program has been widely used throughout the research community. Parallel implementations on both coarse and fine grain machines were reported after 1987. A parallel decomposition introduced by Tanese has been used with little variation in all Hypercube implementations reported in the literature (73:179). In Tanese's decomposition, solutions are shared among genetic algorithms assigned to each node of the Hypercube.

Facial Recognition (10:416-421)
Classifier Systems (68:324-333)
Robot Trajectories (17:144-165)
Resource Scheduling (72:502-508)
NP-Complete Problems (46:231-236)
DNA Conformational Analysis (56:251-281)
Database Query Optimization (3:400-407)
Parametric Design of Aircraft (6:213-218)
Neural Net Architecture Synthesis (45:202-222)

Figure 2. A Few Genetic Algorithm Applications

While sequential and parallel genetic algorithms generally are quite successful at returning "good" solutions[3], difficulties sometimes arise. A problem sometimes associated with both sequential and parallel genetic algorithms is known as "premature convergence." In premature convergence, the search stalls at a locally optimal solution. While in some cases difficulties like premature convergence can be corrected by a change in parameter settings, Bethke and Goldberg have shown that certain classes of problems are intrinsically difficult for genetic algorithms (GA) (4) (32) (33). These difficult problems are known as GA-hard problems. In response to the difficulties standard genetic algorithms have with GA-hard problems, Goldberg, Deb, and Korb have developed a new type of genetic algorithm known as a messy genetic algorithm. Empirical results have shown messy genetic algorithms to consistently outperform standard genetic algorithms when applied to GA-hard problems (36) (37) (38).

A limited amount of research into genetic algorithms has been conducted at the Air Force Institute of Technology (AFIT). In their doctoral research, Lt Col Bruce Conway and Capt Ronald Jackson used Grefenstette's *Genesis* genetic algorithm code to obtain "very good solutions" to a noisy Kohonen net problem and a eigenvalue problem having no closed-form solutions, respectively[4]. Capt George Sawyer parallelized Grefenstette's genetic algorithm using the typical coarse-grained decomposition (71). Additionally, the permutation version of Grefenstette's code was similarly decomposed as part of a special study related to this thesis effort. These parallelized genetic algorithms form the foundation of a planned Genetic Algorithm Toolkit for hypercube computers.

[3]There seem to be several criteria of "goodness" in the literature. The nearness of the solution to the known optimal or relative to the results of other search techniques (especially simulated annealing) seem to be the most common criteria used by researchers, while those using genetic algorithms to solve practical problems base their assessment on how well their solution requirements are met.

[4]Personal interview with Lt Col Jackson

1.3 Problem Statement.

In support of the development of the Genetic Algorithm Toolkit, this thesis effort attempts to address the limitations in the available genetic algorithms, specifically the problems of premature convergence and GA-hard problems.

1.4 Research Objectives.

The research objectives resulted from a decomposition of the broad problem statement mentioned in the previous section into smaller, more tangible goals. The objectives of this thesis effort are

- To gain insight as to why premature convergence occurs.
- To provide guidance and techniques on how to improve solution quality if premature convergence is suspected.
- To extend current capability by designing and implementing sequential and parallel versions of a messy genetic algorithm.
- To compare results for the messy genetic algorithm with the early results of Deb, Goldberg, and Korb.
- To analyze the sequential and parallel versions of the messy genetic algorithm in term of efficiency and effectiveness.
- To provide a foundation for further development of the Genetic Algorithm Toolkit by adequately documenting, analyzing, and validating the design and code developed in the course of this thesis effort.

1.5 Research Questions.

In support of the accomplishment of the research objectives, this research effort addresses the following questions:

1. Is the parallel implementation of the standard genetic algorithm theoretically sound?
2. If the parallel implementation diverges from GA theory, do modifications to improve the theoretical correspondence of the algorithm alleviate the premature convergence problem?
3. What, if any, benefit is obtained by sharing the solutions among genetic algorithms executing on the Hypercube?
4. Do the test results of the sequential messy genetic algorithm developed in this research effort correspond to the published results of Deb, Goldberg, and Korb?
5. Is the messy genetic algorithm amenable to parallelization? (The messy genetic algorithm has yet to be parallelized). Specifically, does the parallel implementation result in a speedup without a loss of solution quality?

These research questions are addressed in the conclusions to this thesis.

1.6 Assumptions.

- *Genesis* Code. Grefenstette's *Genesis* code is assumed to be validated since it has been used successfully for several years. No independent validation of *Genesis* is performed. However, Sawyer's parallel implementation is tested for logical errors (Chapter 3).
- Assumptions on Readership. This research is primarily concerned with parallel genetic algorithms. While a cursory review of sequential genetic algorithms is provided, many details are not included. For a more detailed explanation, Goldberg's book is an excellent source (34). Data flow diagrams, data dictionary, and pseudocode are used to document the design of the

messy genetic algorithm. No tutorial on the associated terminology is provided. Yourdon's book is highly recommended (76).

1.7 Scope.

One problem that has a history of exhibiting premature convergence on a hypercube genetic algorithm is analyzed. The problem is the minimization of the function known as Rosenbrock's Saddle. Methods to alleviate the premature convergence associated with this problem are explored. As is shown in Chapter 2, a genetic algorithm is defined or "instantiated" with numerous parameters that are believed to have a non-linear relationship (40:122-123). Finding optimal settings is a combinatoric search problem in itself. So the parameters and strategies examined in the attempt to reduce premature convergence are limited to the ones judged most likely to offer improvement—population size and selection strategy. Chapter 3 provides further discussion and justification for these choices.

The messy genetic algorithm analysis, design, implementation, parallelization, and validation is an extensive and detailed process. So this initial research into this search process focuses on parallelizing the algorithm, something that has not been done before. To validate the parallel mGA, it is first applied to a classical problem (36:510-511). Ideally, the mGA should be applied to a problem of practical importance (see Chapter IX).

1.8 Limitations.

A problem with a semi-optimal search method such as a genetic algorithm is that it gives no indication of the quality of the solution it returns. In the current state of the art of genetic algorithms, the user of the program is not even guaranteed a "good" solution has been found, only that the solution being returned is the best in the subset of the search space explored.

Consequently, the user must decide whether premature convergence has occurred. To make an immediate decision requires knowledge of the problem domain. Lacking such knowledge makes it necessary for the user to perform some sort of relative evaluation— comparing results with different paramenter settings, use of different search techniques, or comparison with literature results. The ultimate test is whether the solution found adequately meets the user's requirements. Such a decision can only be made by the user.

Another limitation is that the selection of the type genetic algorithm to use—standard or messy—also lies with the user. Again, knowledge of the form of the function governing the problem is important in making this decision. Neither genetic algorithm "flags" the user that the problem the user wants optimized is difficult for it to solve, and that perhaps an alternate strategy should be used. An expert system front end may one day help in choosing the appropriate genetic algorithm, but such a capability is not currently provided.

In using the messy genetic algorithm, the building block size (non-linearity of the problem) must be input by the user. Generally, the non-linearity of the problem is not known. In tests conducted with the messy genetic algorithm developed in this research, if the "wrong" non-linearity is input, solution quality is negatively affected.

Finally, the Genetic Algorithm Toolkit is not yet integrated. Currently, each genetic algorithm has its own user interface and resides in different directories. Non-uniform interfaces and the need to manually change directories detract form user friendliness. Ideally, the Toolkit should have a single interface allowing the user to choose the desired genetic algorithm.

It should be pointed out that the limitations mentioned here (except the user interface) reflect the state-of-the-art in genetic algorithms. Hopefully, the limitations will not daunt anyone from using genetic algorithms. A review of the literature reveals numerous success stories in diverse applications.

1.9 Expected Benefits of This Research.

A goal of the Genetic Algorithm Toolkit is to provide users with a robust search package. Addressing the problem of premature convergence and adding a messy genetic algorithm for GA-hard problems are steps towards meeting this goal (Figure 3). With the addition of a better user interface, AFIT will have an excellent genetic search package which could benefit all disciplines within the school.

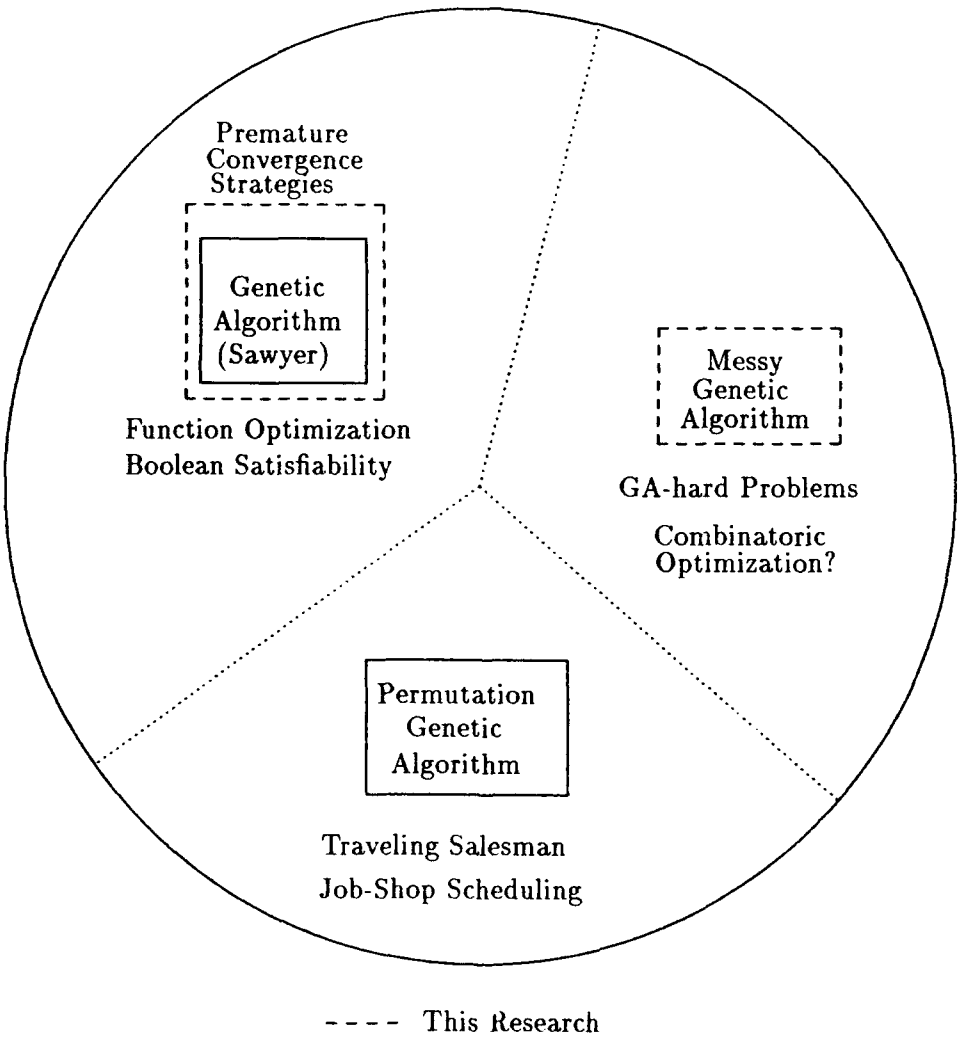


Figure 3. AFIT's Genetic Algorithm Toolkit (Current Status)

1.10 Summary.

This chapter began with a general introduction of search, emphasizing the problems of inefficiency and lack of robustness in many search techniques. It was indicated that genetic algorithms, a class of stochastic search techniques, have shown great potential to efficiently search broad domains, motivating the development of a Genetic Algorithm Toolkit in this research. A background discussion summarized several milestones in the development of genetic algorithms, including past uses and current capability at AFIT. Since the current capability was found to be lacking in providing strategies for dealing with premature convergence and GA-hard problems, research objectives and questions were formulated to address these deficiencies. The assumptions, scope, limitations, and expected benefits of this thesis effort were also provided.

1.11 Layout of Thesis.

The remainder of this thesis is organized as follows. Chapter II contains the literature review, which focuses on parallel implementations of genetic algorithms and GA-hard problems. Chapter III discusses and justifies the methods used to examine and reduce premature convergence in a target problem. Chapter IV contains the experimental results. Chapter V discusses the methodology and design of a sequential messy genetic algorithm. Chapter VI details the steps taken to parallelize the messy genetic algorithm. Chapter VII contains the results obtained from parallelizing the messy genetic algorithm. Chapter VIII discusses the conclusions derived from this research. Chapter IX offers recommendations for the project and for future research.

II. Literature Review.

This chapter contains the literature review conducted in support of this thesis effort. Section 2.1 summarizes Holland's work in the abstraction of the salient features of natural adaptive systems. Section 2.2 describes how the abstractions are generally implemented in a genetic algorithm. Section 2.5.1 gives an overview of past research in parallel genetic algorithms, in particular Hypercube implementations. Section 2.4 introduces problems that are inherently difficult for simple genetic algorithms, the GA-hard problems.

2.1 An Abstract Model of Adaptation.

In Chapter 1, efficiency, effectiveness, and robustness are cited as desirable characteristics of an optimizing search algorithm. The correspondence between characteristics of a successful search strategy and a successful species in nature is intuitively appealing. A species (solution) adapts to its environment (problem), becoming better suited (more optimal) to the environment as time goes on. A successful species adapts to changes in the environment. That is, a species must be robust to survive. If the adaptation to a environment occurs too slowly, the species may die out. So a certain amount of efficiency seems a requirement. Observations such as these motivated the development of genetic algorithms. Bridging the gap between intuition and algorithm development was not a trivial step, however.

Much of the early development of genetic algorithms was the work of John Holland of the University of Michigan. His book, *Adaptation in Natural and Artificial Systems* (47) describes how structures and operations similar to those found in natural systems can be used in an optimizing search. To take advantage of the characteristics of a successful species in a search process, Holland identified and abstracted the salient features of an adaptive system in nature.

The identification of the key components of a natural system system allowed Holland to construct an abstract model that could be applied to artificial processes. To allow a formal, un-

ambiguous description of an adaptive system, Holland used symbols to represent its components (47:20-31). Key components and their associated symbols are an adaptive plan τ, structures $\mathcal{A}$, and a set of operators ω. The adaptive system evolves in response to signals I that are generated by the environment E. Figure 4 defines the key components in an adaptive system.

Symbol	*Definition*
E	environment to which a system is adapting
I	set of inputs to system from environment
$\mathcal{A}$	the domain of possible structures
ω	set of operators for modifying structures
τ	adaptive plan–specifies how operators are applied (algorithm)

Figure 4. Holland's Symbology (47:28-29,171)

Holland's next step was to identify the relationships between the adaptive components. His symbology allowed the relationships to be described formally using equations (47:21-25). The relationships indicated by his equations were used to construct Figure 5. Notice the adaptive process involves interaction between the adaptive system and the environment, as well as interactions between the components of the adaptive system.

2.2 Genetic Algorithm Development.

The abstract model presented in the last section is quite general. Further refinement was needed before a genetic search program could be developed. However, no specific data structures, operators, adaptive plan (algorithm) were mandated as being essential by Holland. His initial description is very general (47:1-19), allowing "considerable latitude" in the choice of structures (47:159). Evidently, an infinite number of data structures and operators could be used to form a genetic program. Despite this flexibility, Holland again made the logical choice and let nature be his guide. His selection of fairly simple structures and operations facilitated a rigorous description of the behavior of a restricted class genetic algorithms (see Section 2.3.1).

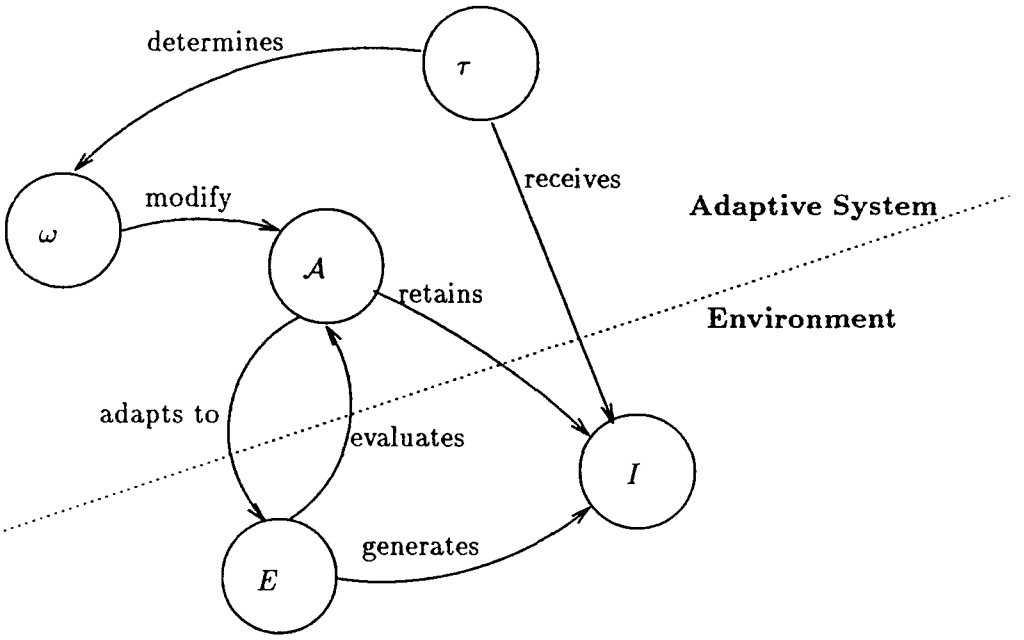


Figure 5. Concept Map of an Adaptive System

2.2.1 Data Structures. The structures Holland developed are quite similar to chromosomes. While Holland describes some fairly complicated chromosomal structures and operators, the bulk of his theoretical work focused on a single-stranded, constant-length string (l-tuple) (47:89). As the terminology associated with the composition of strings and chromosomes is often used interchangeably, Figure 6 provides a semantic map.

chromosome	string
allele	value
locus	position
gene	(position, value)

Figure 6. Correspondence Between a Chromosome and a String

The string data structures Holland describes in *Adaptation in Natural and Artificial Systems* have been widely used in practical applications of genetic algorithms, despite the fact that other, more complicated structures are not prohibited (47:141-158). The theoretical foundation, simplicity,

and generally excellent results in broad problem domains[1] associated with fixed-length strings are likely reasons why other structures are seldom used.

In an optimization problem, the string is an encoding of the parameters which specify a solution to the problem (34:7). The length of the string is dependent on the number of parameters, the domain of the parameters, and the desired resolution (precision). Characters from an alphabet are used to encode the string. While any alphabet with a cardinality greater than two could encode the parameters of a problem, Goldberg recommends the use of "the smallest alphabet that permits a natural representation of the problem" (34:80). Strings encoded with smaller alphabets allow a more efficient search than strings encoded with larger alphabets (34:82). So a binary encoding seems quite desirable.

For some applications, the encoding of the parameter set to binary form is fairly straightforward and natural. For example, the x and y values for a function f(x,y) might be encoded as binary string of length $2n$, with the first n bits representing the x parameter and the second n bits representing the y parameter. A binary encoding such as this is typically used in functional optimizations problems. A fixed length binary string also is a natural representation of the Boolean Satisfiability Problem, an important NP-Complete problem (20:173). For ordering problems such as the Traveling Salesman Problem and job-shop scheduling, a binary representation is less natural than a decimal encoding (20:171) (43) (11:160-169), but can still be used (61:474-479).

2.2.2 The Population. To broaden the search, genetic algorithms operate on a "population" consisting of multiple strings (34:9). Goldberg's analytical work on binary coded strings suggests that the optimal size of a population is a function of the string length (30). In most applications of genetic algorithms, the size of the population remains fixed (14:148) (41), possibly due to the resulting simplicity of the data structure (a static array). However, neither Golderg's nor Holland's theoretical results prohibit the use of variable-sized populations.

[1] As is discussed shortly, GA-hard problems are an exception

2.2.3 Genetic Operators. Genetic operators (ω) operate change the composition and distribution of the population of solutions. Some of the more common common genetic operators are described in the following paragraphs.

2.2.3.1 Evaluation. The evaluation operator assigns a "fitness" to a string which is "some measure of the profit, utility, or goodness" of the string in terms of how well it solves the problem being optimized (34:1). Formally, fitness is a component of the input I (47:25). In all applications reviewed in the literature, the fitness is the only component of I, meaning that the genetic algorithm uses only a "black-box" view of the problem. Since genetic algorithms generally use no internal or heuristic information related to the problem, such as subtour lengths in a Traveling Salesman Problem, it is often said that they perform a blind search.

For a function maximization problem, the fitness of a string S is often just set equal to f(S*), where S* is the set of the decoded variable values and f is the function (34:11). However, assignment of fitness is less straightforward for functional minimization and many NP-complete problems, since the goal is to find a solution which minimizes the value of the objective function. In such cases, a mapping must be performed which assigns high fitness values to low objective values. One such mapping is to simply subtract the functional value from an arbitrary "high" value. For instance, in Grefenstette's *Genesis* algorithm, fitness is assigned by subtracting the functional value from the the highest functional value encountered in the last N generations, N being a user-specified value. (41).

In other problems, assigning fitness is somewhat contrived. In the Boolean Satisfiability Problem, for example, either the boolean expression is satisfied or it is not. Having just two fitness values "would not allow the formation of useful intermediate building blocks" needed to find an optimal solution. To form a "fitness gradient" in a Boolean Satisfiability Problem, one approach has been to assign fitness based on how many conjunctive clauses are true (21:127-128). For the Set Covering Problem, another problem often without a natural fitness gradient due to the cover

constraint, an "artificial" fitness gradient can by created using a penalty function for failure to cover (55:91-92).

2.2.3.2 Selection. The selection operator changes the frequency distribution of a population of strings based on fitnesses of the strings. In doing so, the selection operator implements the concept of "survival of the fittest." In his theoretical work, Holland used proportional selection, in which the expected number of populations slots allotted to a string is directly proportional to the string's relative fitness (42:20-21). An element of non-determinism is typically introduced into the selection mechanism. To implement non-determinism in proportional selection, Goldberg uses a roulette wheel in which the number of slots assigned to a string is directly proportional to fitness (34:11). To reproduce a population of size N, the wheel is spun N times (34:34). Since poor solutions are given proportionately fewer slots, their numbers will likely be reduced as the result of selection, while good solutions are likely to receive multiple copies in the next population (34:10-12).

Several other selection algorithms have been introduced, none of which has been shown to be consistently better than other selection algorithms in terms of solution quality (42:25). However, in his survey of selection algorithms, Baker found differences in terms of the time efficiency. For example, roulette wheel sampling is typically implemented in $O(N^2)$ time[2], while some of the newer selection algorithms have $O(N)$ time complexity (1:15,19). One of these $O(N)$ selection algorithms, known as "Stochastic Universal Sampling" (SUS) algorithm, is used in the *Genesis* genetic algorithm employed in this research (41). While SUS is classified as a "strictly sequential" algorithm by Baker (1:16), a modification allows it be be effectively parallelized (see Chapter 3).

Unlike selection, which simply redistributes population slots among existing strings, the remaining genetic algorithms presented here produce new strings. Since a new string represents a new

[2] $O(NlogN)$ using a B-tree

solution, these "recombination" operators offer the potential for improvement in solution quality (47:13).

2.2.3.3 Crossover. In crossover, strings are paired up as mates, and at a random location their gene values (alleles) are exchanged. Crossover is typically implemented as a stochastic operator whose rate is controlled by the crossover probability, a user-specified value. The crossover probability specifies the likelihood that the two mates undergo crossover. If a crossover does occur, two new solutions (children) may be produced as shown in Figure 7 (34:12-13).

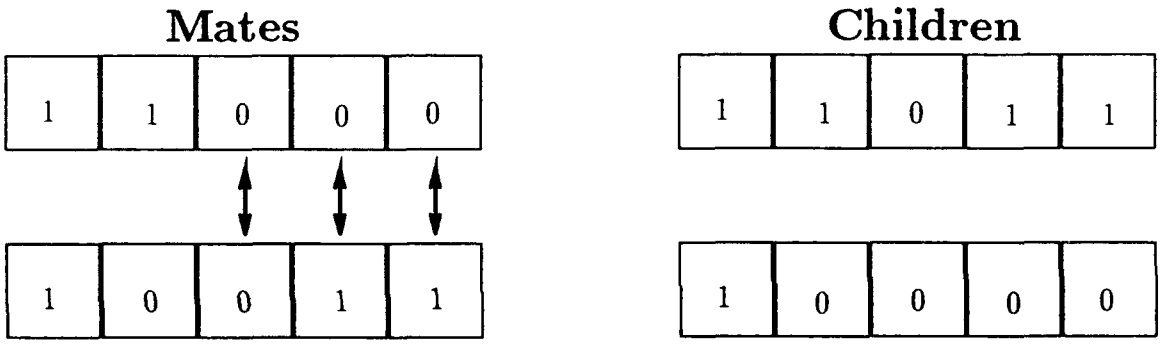


Figure 7. Crossover Creates New Solutions

The effectiveness of crossover in producing new solutions is dependent on the diversity of the population. If the mates have several differences in their bit patterns, there is a good chance that new regions of the search space will be examined as the result of crossover. Since quite a substantial "jump" may occur from the "parent" strings to the "child" strings, nothing like a tree or graph control structure is used with a genetic algorithm.

Notice, though, if the portions exchanged are identical, neither solution changes. In terms of a search, crossover which does not yield new solutions is very undesirable. When a population has converged (all solutions nearly identical), the search stagnates.

2.2.3.4 Mutation. Mutation, another stochastic operator, helps to maintain diversity in the population by altering allele values of the strings in random locations. By allowing genes that have disappeared from the population reappear, mutation provides an "insurance policy against premature loss" of an allele value (73:181). For example, say a population of binary strings evolves so that all strings in the population have a 1 in the third position of the string. Using crossover alone, a 0 never again occurs in the third position, so a portion of the search space essentially becomes "closed off" to further search.

However, mutation often cannot provide a solution to premature convergence. Mutation is a very disruptive operator. In a population of "good" solutions, a mutation is very likely to produce a more inferior solution. To avoid an essentially random search (23:10), mutation rate is typically set to a very low values (.001 is a typical value) (18:15).

2.2.3.5 Inversion. Inversion is a unary operator that inverts the bits between two random locations in a string (34:21). An inversion operator requires a more complex data structure since the position must be inverted as well as the bit value (34:166-167). Due to the overhead associated with inversion, it is rarely used in practice. It is only mentioned here because it played a part in Holland's proofs of robustness and efficiency.

2.2.3.6 Other Operators. Several other genetic operators have been developed (34:147-208). For example, since the standard crossover operator tends to produce invalid tours in the Traveling Salesman Problem (TSP), a new crossover operators such partially matched crossover were developed which guarantee valid tours (39:154-156). Since ordering problems like the TSP are not considered in this thesis effort, and the other operators are seldom used, they are not considered further.

2.2.3.7 Effect of the Application of the Operators. Notice that crossover and mutation have orthogonal roles within a genetic algorithm. Whereas crossover and mutation tend to increase

diversity by changing the bit patterns, selection tends to increase uniformity by giving more population slots to fitter strings in the next "generation." The combined effect of the operators is a search process that allows for improvement by searching new regions of the solution space, but which also focuses on regions offering the greatest promise (47:13-16).

To arrive at a "good" solution in a reasonable amount of time, it seems a careful balance has to be maintained. If crossover and mutation dominate, "natural selection" could be subverted as the search becomes overly random. If the selection dominates, a only a limited portion of the search space may be searched before the population becomes too uniform to produce better solutions. Control of the "trajectory" of the genetic algorithm falls under the domain of the adaptive plan, discussed next.

2.2.4 The Adaptive Plan and Algorithms. An adaptive plan specifies a set of genetic operators (47:121-123). Subordinate to an adaptive plan are subclasses which specify how (e.g., in what order) the genetic operators are to be applied. Each subclass is a genetic algorithm (47:122-123). As there are numerous variations in the operators and their application, there are numerous genetic algorithms. Hence the plurality in the name of this field of study.

One genetic algorithm, called by Goldberg the simple genetic algorithm (34:59), seems to be the most commonly used. The simple genetic algorithm has performed effectively against problems of practical import (34:10) through the use of only reproduction (evaluation and selection), crossover and mutation[3]. DeJong used a simple genetic algorithm in his important research on functional optimization (19). An examination of Grefenstete's *Genesis* genetic algorithm, a popular genetic algorithm package (41), shows it to be a simple genetic algorithm.

In *Genesis* genetic algorithm, the genetic operators are applied as follows (41):

1. Initialize the population of strings. Random initialization is typically used.

[3]Since Goldberg does not specify an algorithm, only the use of a set of operators, he seems to be talking about an adaptive plan rather than an algorithm. As the same set of operators may be applied in various ways, each way being a new algorithm, a single adaptive plan has many algorithms as subclasses.

2. Evaluate the strings in the population.

3. Perform selection on the population.

4. Apply crossover with a probability specified by the crossover rate to the entire population.

5. Apply mutation with a probability specified by the mutation rate.

6. If the some termination criteria is met, exit. Otherwise, go to step 2.

One iteration of the evaluate-select-crossover-mutate loop is called a generation. A survey of the literature shows by far the most common termination criteria is the completion of a user-input number of generations.

2.2.4.1 Implementation Adjustments to a Genetic Algorithm. An interesting thing to note is that in a genetic algorithm, the goal appears to maximize the average population fitness rather than individual fitness. In fact, due to the action of the genetic operators, "the best individual encountered so far may not even survive into the next generation" (20:170). This seems contrary to the goal of an optimizing search. So while no attention is paid to the best solution in the algorithm above or Holland's adaptive plans (47:92, 95), in practice a the genetic algorithm should include a test in each generation for the best overall solution. Such a test for the best is included in Grefenstette's *Genesis* genetic program (41).

2.2.4.2 Controlling Genetic Algorithms. The algorithm presented in the last section must be "instantiated" before it is executed. That is, various parameters such as the crossover rate, mutation rate, and population size must be specified. Adjusting these parameters influences the behavior of the resulting genetic program. While varying parameters such as crossover and mutation rate often provides some means of alleviating too great a focus or too much randomness, apparently an effective balance cannot always be achieved: "Premature convergence—loss of population diversity before before optimal or at least satisfactory values (solutions) have been found—has long been recognized as a serious failure mode for genetic algorithms" (24:115).

2.3 *Behavior of Genetic Algorithms.*

2.3.1 Holland's Theoretical Results. Problems such as premature convergence did not seem to be predicted by Holland's theoretical analysis of genetic algorithms. While the details of the Holland's proofs will not be presented here, the results are quite interesting. Holland showed there is "implicit parallelism"[4] associated with a genetic algorithm. In general terms, "implicit parallelism" is "the simultaneous allocation of search effort to many regions of the search space according to sound principles" (42:20). Implicit parallelism results from the independent processing of schema in the population of strings (47:71-72). A schema is a "similarity template describing a subset of strings with similarities at certain positions" (34:19). For example, the schema *111* represents a set of four strings, {01110, 01111, 11110, 11111}. Holland's Schema Theorem asserts that his particular genetic algorithm propagates "highly fit, short-length schemata (building blocks) from generation to generation by giving *exponentially* increasing samples to the best" building blocks (34:20), while below average schema "rapidly decline in numbers" (42:21).

Holland also showed that the Schema Theorem remains in effect in a genetic algorithm employing proportional selection, crossover, inversion and variable mutation, regardless of the complexity of the fitness function (47:129). In this sense, he showed this specific class of genetic algorithms to be robust (47:121-140).

2.3.2 Empirical Results. Interestingly, a simple genetic algorithm does not belong to the class of algorithms used by Holland in this robustness proofs. Looking again at Section 2.2.4, we see the simple genetic algorithm does not have an inversion operator. In fact, no actual genetic algorithm surveyed in the literature uses the specific algorithm (47:122) on which Holland based his proofs. This does not necessarily mean the genetic algorithms used in practice are neither robust nor efficient, only that they have yet been proven to be so. Grefenstette and Baker have attempted to

[4] This is not the type of parallelism that could be directly exploited by a parallel computer.

extend Holland's theoretical results to practical implementations of genetic algorithms, yet further effort is needed to "characterize the behavior of good genetic algorithms" (42:26).

Fortunately, most empirical evidence attests to the effectiveness of genetic algorithms. Baker performed extensive testing on a class of genetic algorithms he called *reliably consistent* (RC) genetic algorithms (42:25), a class which seems to include many of the genetic algorithms reported in the literature. As the class of RC genetic algorithms use crossover, mutation, and various forms of selection, they include the simple genetic algorithm. Applying RC genetic algorithms to several functional combinatoric and functional optimizations problems, Baker "shows that for any such genetic algorithm, empirical tuning of the parameters defining the selection method yields an algorithm with high performance" (42:25) (2). GA-hard problems seem to give genetic algorithms difficulty, however.

2.4 GA-hard (Deceptive) Problems.

In his doctoral research, Albert D. Bethke demonstrated that a simple genetic algorithm is not well-suited for certain functions (4). The class of functions he studied are known as Walsh functions, of practical importance since "any real-valued function defined on bit strings ...can be written as a Walsh polynomial" (26:188). Bethke formulated Walsh functions having misleading schemata (building blocks) that drove the simple genetic algorithm to sub-optimal solutions (4:78-90). Several other researchers have studied and formulated functions inherently difficult for the simple genetic algorithm (31:74-88) (74:434-440) (16:166-173) (26:182-189) (48:196-203). These functions have come to be called GA-hard or deceptive. Use of inversion to improve performance against GA-hard problems was found to be inefficient (8:397-405).

To deal more effectively with GA-hard problems, a new type of genetic algorithm, known as a messy genetic algorithm, was developed by Kalyanmoy Deb of the University of Alabama, Bradley

Korb of McDonnell Douglas Space Systems, and David Goldberg of the University of Illinois. From an evolutionary perspective, the differences between the two algorithms are as follows:

- Simple Genetic Algorithms. Short term evolution. All solutions are of the same species. That is, the solutions have the same number of genes. The strings are also completely specified in that each is a complete solution to the problem, having no redundant or contradictory alleles. Evolution is concerned with making solutions increasingly optimal.
- Messy genetic algorithms. Long-term evolution. The starting population consists of "building blocks" that underspecify the problem. Genetically, building blocks are a "lower form of life" than a a fully-specified solution. The genetic operators act to create solutions that more fully specify the problem.

These difference translate to substantial differences in terms of data structures and operators (Figure 8). A more detailed explanation of the terminology and operation of the messy genetic algorithm is contained in Chapter 3, but such radical differences between the two algorithms suggest little opportunity for design or code reuse in developing a messy genetic algorithm.

	Simple Genetic Algorithm	Messy Cenetic Algorithm
String Data Structure	Fixed Length	Variable Length
Genetic Operators	Crossover, Mutation, (sometimes) Inversion	Cut and Splice
Evolutionary Phases	Generations	Primordial, Juxtapositional
Population Size	Constant	Variable
Initial Population	Fully-specified	Under-specified

Figure 8. Some Differences between Simple and Messy GAs

The preliminary results suggest the effort is justified. A sequential LISP version of a messy genetic algorithm returned the optimal solution for several GA-hard functions, while the simple genetic algorithm returned the sub-optimal results (38:28-29). The researchers have yet to devise a

proof of convergence to optimality, but their promising early results have caused them to recommend the application of messy genetic algorithms to problems of practical importance (38:28-29).

Before attacking large combinatoric optimization and other practical problems, parallelizing the code would be very desirable. A full parallel implementation would have logarithmic time complexity, a significant improvement over the sequential algorithm with its polynomial time complexity (38:27). Additionally, the strain put on the memory resources by a messy genetic algorithm could be alleviated with a parallel implementation (Deb, Goldberg, and Korb had to resort to the use of subpopulations when memory resources were exceeded (37:438)). AFIT's iPSC/2 Hypercube has over 80 Mb in total node heap space, much greater than the memory resources on any workstation or mainframe computer at AFIT.

2.5 Parallel Genetic Algorithms

Most of the discussion until now has dealt with how well genetic algorithms perform in terms of solution quality. Efficiency is an important consideration as well. Despite the non-exponential time complexity of genetic algorithms, at times they cannot meet time limitations against problems requiring large population sizes (73:177) or having complex evaluation functions (63:155). To reduce the execution time of genetic algorithms, several researchers examined parallel implementations of genetic algorithms. Since parallel implementations of genetic algorithms differ from their sequential counterparts, often different solutions are obtained. Fortunately, in most cases the solutions are at least as good as those returned by the sequential version.

2.5.1 Past Research in Parallel Genetic Algorithms. Goldberg gives an excellent summary of the many diverse application areas in which genetic algorithms have been used (34:127-129). The parallel implementations Goldberg mentioned, together with several implementations reported since the publication of his book, are shown in Figure 9.

Year	Investigators	Description
Parallel Genetic Algorithms		
1976	Bethke	Brief theoretical investigation of possible parallel GA implementations
1981	Grefenstette	Brief theoretical investigation of several parallel GA implementations
1987	Cohoon, Hegde, Martin, and Richards	Simulated parallel implementation of optimal linear arrangement
1987	Jog and Van Gucht	Combined knowledge-based and parallelized GA
1987	Pettey, Leuze, and Grefenstette	Parallel GA on Intel hardware using De Jong test bed
1987	Suh and Van Gucht	Localized selection in parallel GA search on the TSP
1987	Tanese	Parallel GA implemented on 64-node NCUBE computer
1989	Robertson	GA on Connection Machine used for t[illegible]olution of rules for letter sequence prediction
1989	Pettey and Leuze	Showed that sending a random individual to nearest neighbors every generation maintains the efficiency associated with sequential GAs.
1989	Brown, Huntley, and Spillane	Hybrid of GA and simulated annealing (SA) used on the nodes of a Hypercube to solve the Quadratic Assignment Problem (QAP)
1989	Muhlenbein	GA implemented on transputer system gave best solution known for the largest published QAP
1989	Gorges-Schleuter	GA implemented on transputer system using ladder topology gave better solutions than SA for the TSP
1989	Manderick and Spiessens	Found no optimal population size for a GA implemented on a Connection Machine
1989	Tanese	GA on a Hypercube gave a near linear speedup and better solutions for Walsh functions than a sequential GA
1991	Cohoon, Martin, and Richards	Obtained linear speedup on a Hypercube using GA with local selection for the K-Partition problem
1991	Collins and Jefferson	Local selection and mating found more adept at finding multiple optimal solutions than global selection/mating
1991	Davidor	Proposed a torus topology to promote quick convergence of subpopulations
1991	Husbands and Mill	GA for scheduling problem having string for each component job rather than entire schedule (typical) to take fuller advantage of parallelism
1991	Muhlenbein, Schomisch, and Born	"Superlinear" speedup in terms of time to find best solution for a GA using local hill-climbing on a MEGAFRAME HyperCluster
1991	Spiessens and Manderick	Linear time complexity of fine grained algorithm made large populations feasible

Figure 9. Applications and Implementations of Genetic Algorithms

2.5.2 Hypercube Implementation Details. All Hypercube implementations of genetic algorithms reported in the literature use a nodal algorithm similar to the following (73:179):

```
BEGIN
    Generate initial population
    For gen = 1 to G loop      ; G = maximum generations
        Evaluate fitness of solutions
        Reproduce population
        Perform crossover based on crossover rate CR
        Perform mutation based on mutation rate MR
        IF (gen mod E = 0) THEN      ; E = Epoch (generations between exchange)
            Choose solution(s) for exchange with other processors
            Send solutions to other processors
            Receive solutions from other processors
            Integrate received solutions into population
        END IF
END
```

2.5.3 Analysis of the Hypercube Implementation.

2.5.3.1 The Decomposition. Looking at the decomposition, one sees that these Hypercube implementations are not really a single genetic algorithm. The reason for saying this is these implementations have multiple, virtually independent genetic algorithms running in parallel. So in fact, there is no decomposition of the genetic algorithm into parallel components, but rather an integration of separate genetic algorithms into a meta-type algorithm. None of the operations involved in a sequential genetic algorithm are truly parallelized (Figure 10).

2.5.3.2 The Population. Using the decomposition above allows the simultaneous search of n populations of size M (where n is the number of nodes in the Hypercube and M is the population size used in the sequential version). Since n solutions from n genetic algorithms can be compared and the best selected, there is a definite potential for better solutions than from a single genetic algorithm. However, depending on the amount of communications overhead, the parallel version might be slower than the sequential version. Here is definitely a different mind-

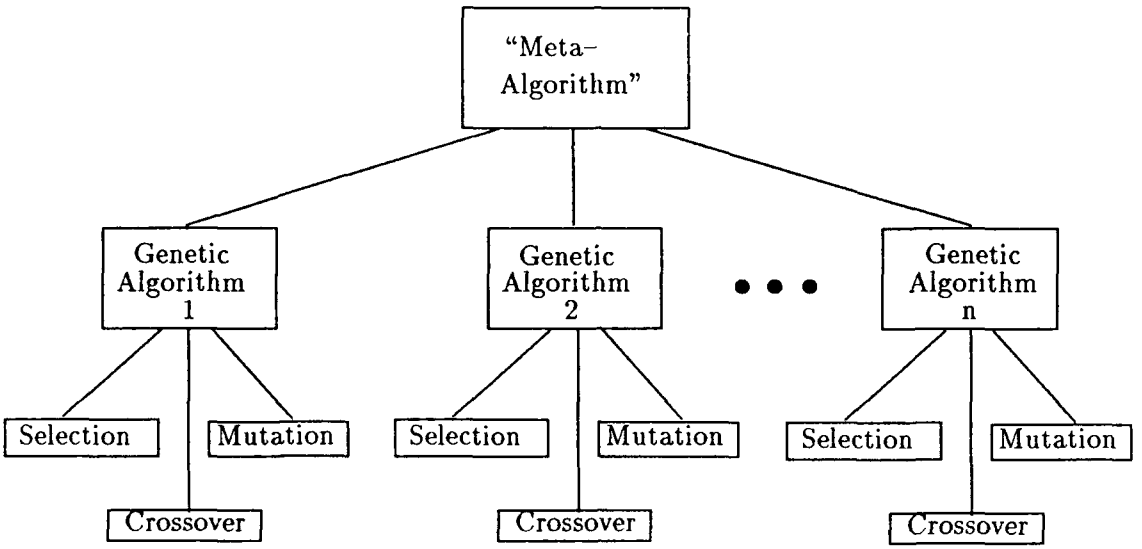


Figure 10. Typical Hypercube "Decomposition" of Genetic Algorithm

set—normally the goal of parallelizing an algorithm is to allow solution of the problem in a lesser amount of time.

However, most users of Hypercube genetic algorithms attempt to achieve speedup. For example, Tanese distributed the population among the nodes of the Hypercube, giving each node a population size of M/n (73:180). With this data decomposition, Tanese achieves near-linear speedups and "comparable performance" in terms of solution quality (73:181).

Tanese's results are unsettling in that Goldberg has put forth analytical arguments suggesting there is an optimal population size for binary-coded genetic algorithms (30). Recall that the genetic algorithms executing on the nodes of the Hypercube are essentially sequential genetic algorithms with a communications step added. It seems logical that the population size that is optimal for the sequential implementation would be the optimal population size for each nodal genetic algorithm. If this is true, using a nodal population size equal to the optimal sequential population size divided by the number of nodes (that is, a non-optimal population size) would result to inferior solutions.

Adding to the confusion is that experimental results using a similar Hypercube implementation on the De Jong functions "corroborated"[5] the theoretical predictions (63:156).

2.5.3.3 Premature Convergence. However, use of sub-optimal population size is not mentioned as a cause of premature convergence when it occurs in a Hypercube genetic algorithm. Instead, one researcher cites the selection strategy while cites the communication strategy as possible causes. Pettey points out that in a parallel genetic algorithm, selection is carried out on a local (nodal) basis, rather than a global basis. Such a local selection strategy has "no theoretical basis" (63:156). Tanese notes that the solutions that are exchanged between the nodes are in effect given more offspring than selection alone would allow (73:179). As a result, there is a greater potential for domination of a population by a locally optimal solution (premature convergence) (73:179).

The parameter settings, communication strategies, and speedup reported for various coarse-grain implementations reported in the literature are shown in Figure 11. The wide variation in parameter settings and strategies is indicative of the high degree of empiricism associated with practical applications of genetic algorithms. NN represents communication with nearest neighbors. Unreported information is indicated with a "?".

Citation	Problem	Standard Parameters				Dim	Communications		Speedup
		G	CR	MR	P	(d)	E	Details	
(14:148-154)	Optimal Linear Arrangement	100-300	0.5	0.05	80	2	2 and 50	15 solutions to NN	Linear
(63:155-161)	DeJong's functions	100	?	?	50	0-4	1	Best w/NN	?
(73:177-183)	Walsh function	1000	0.6	0.008	$400/2^d$	0-6	2 and 5	10-20% of pop. with 1 NN	Near Linear
(64:398-405)	DeJong's functions	100	?	?	50	3	1	1 random with NN	? Linear
(13:244-248)	Partition Problem	1500	0.5	0.1	80	4	50	9 random w/NN	Linear
(60:271-285)	Functions	?	0.65	0.3	20	2,3,6	10	Best w/NN	"Super-Linear"

Figure 11. Details of Coarse-Grain Implementations

[5]De Jong function f2 (Rosenbrock's saddle) was the exception.

2.6 Directions Taken Based on Literature Review.

As the goal of the Genetic Algorithm Toolkit it to provide a robust search package. Currently, only the standard genetic algorithm is included in the Toolkit[6]. As a result, the Toolkit is subject to the limitations associated with the simple genetic algorithm. One limitation is that both simple and parallel simple genetic algorithms have show the occasional tendency to prematurely converge on inferior solutions. Additionally, GA-hard problems are inherently difficult problem for simple genetic algorithms.

To broaden the scope of the Toolkit, the premature convergence and GA-hard problems should be addressed. As premature convergence in parallel genetic algorithms has not yet received much attention, and messy genetic algorithms have yet to be parallelized, research in these areas was thought to be worthwhile.

[6]The permutation genetic algorithm has been parallelized, but not been tested.

III. Methodology - Examination and Reduction of Premature Convergence.

3.1 Introduction.

This chapter describes and justifies the methodology used to address premature convergence in a problem of practical importance. Section 3.2 discusses the function known as Rosenbrock's saddle, and justifies its use in research concerned with finding ways of reducing premature convergence. To gain insights into the problem and the operation of the genetic algorithm , some preliminary testing was performed. The results of the tests and their analysis are presented in Section 3.3. Section 3.4 offers arguments suggesting the use of global selection on a Hypercube as a means of alleviating premature convergence. Section 3.5 presents the steps taken in implementing global selection.

3.2 Target Problem - Rosenbrock's Saddle.

3.2.1 Justification of Choice. Rosenbrock's Saddle, also known as De Jong function f2, has long been used as a litmus test of genetic algorithm performance (19) (34) (63). This function was originally developed by H. H. Rosenbrock for his study of gradient-based step search techniques (69). Rosenbrock's saddle was chosen as the test problem for the following reasons:

- Premature convergence was observed when Rosenbrock's saddle was applied to a Hypercube implementation of a genetic algorithm (63:156-158). While premature convergence is generally said to be a problem with genetic algorithms, this is the only tangible instance documented in the literature.
- Pettey offers no discussion as to why Rosenbrock's saddle is a difficult problem for genetic algorithms. In particular, is Rosenbrock's saddle GA-hard? Other researchers have since examined the search space of Rosenbrock's saddle (24:120), offering reasons why such a function is difficult for a genetic algorithm (see next section), but apparently not GA-hard.

- Pettey offers no suggestions on how the observed premature convergence might be alleviated. He does point to local selection mechanism as a possible cause of the premature convergence (63:156).
- In a study directed at reducing premature convergence in sequential genetic algorithms, Rosenbrock's saddle was the lone function in a suite of test functions whose premature convergence was not alleviated using a strategy that prevented similar solutions from mating (24:115-122). As a result, further study on this function was urged (24:120).
- Pettey sized his populations close to Goldberg's theoretical recommendations (63:156) (30). While the results of the other functions he examined "seemed to corroborate Goldberg's theory," the optimal solution to Rosenbrock's saddle was not found in a range of populations sizes that included the theoretical optimal (63:156-158). An experiment determining the optimal population size for Rosenbrock's saddle, if such an optimal exists, would be useful from a theoretical standpoint.
- Rosenbrock's saddle has practical importance. Rosenbrock's saddle belongs to a class of functions typically associated with the control systems used in air-to-air missile guidance, tracking, and oscillation control in aircraft (57). Considered a very difficult function to optimize, it is routinely used as a standard against which to judge the performance of gradient-based search using the penalty function approach. If a search strategy performs well against Rosenbrock saddle, it is very likely it will perform well against all functions in this class (57). Hence the motivation to improve the performance of the genetic algorithm against this problem.

3.2.2 Description of Rosenbrock's Saddle. Rosenbrock's saddle is a non-linear, non-convex function of 2 variables, having a deep parabolic valley:

$$f2 = 100 * (x^2 - y)^2 - (1 - x)^2$$

Figure 12 shows a linear plot of the saddle. The inverted log plots (Figures 13 and 14) and contour map (Figure 15) of the function show the depth of the "valley."

The first term of the function is completely symmetric about the x-axis, having a minimum of at points satisfying the relation $y = x^2$. The second term, which is much less in magnitude than the first term, introduces slight differences in the functional values along the parabola (Figure 16). Since the second term has a minimum of zero only at the point x = 1, the composite function has one minimum at the point $x = 1$, $y = 1$. The differences in the values along the parabola will not be very large, especially relative to the maximum values (poor solutions).

The function is continuous, having an infinite number of solutions. To allow for a solution by a genetic algorithmic, the solution space must be parameterized. For this study, the genetic code consists of an x- and y-substring parameterized so their resolution is 0.001 and their range is from -2.048 to 2.047. To represent the 4096 numbers in this range, a 24-bit binary string is used, with 12 bits ($2^12 = 4096$) for each parameter.

3.2.3 Why is Rosenbrock's Saddle Difficult for a Genetic Algorithm? Looking closely at the function gives one some insight as to why a genetic algorithm might prematurely converge. While a total of 2^{24} solutions exist in parameterized search space, there are only 2000 points along the parabola. That is, 2000 sets of x and y satisfy the relation $y = x^2$, meaning just $2000/(2^{24}) = 0.012$ per cent of the points are along the parabola. It is unlikely one of the parabolic points will be generated in the initial random generation of solutions. But notice the steep fitness gradient leading to the parabola. Such a clear increase in fitness makes it likely the genetic algorithm will find one or more points along the parabola. Since the points on the edge of the parabola are so superior relative to other points, once one is found it will dominate the genetic population unless even better solutions are quickly found.

Once on the saddle, though, the genetic algorithm is likely to have difficulties finding the global minimum. Genetic algorithms tend to be weak at localized searches since the genetic operators

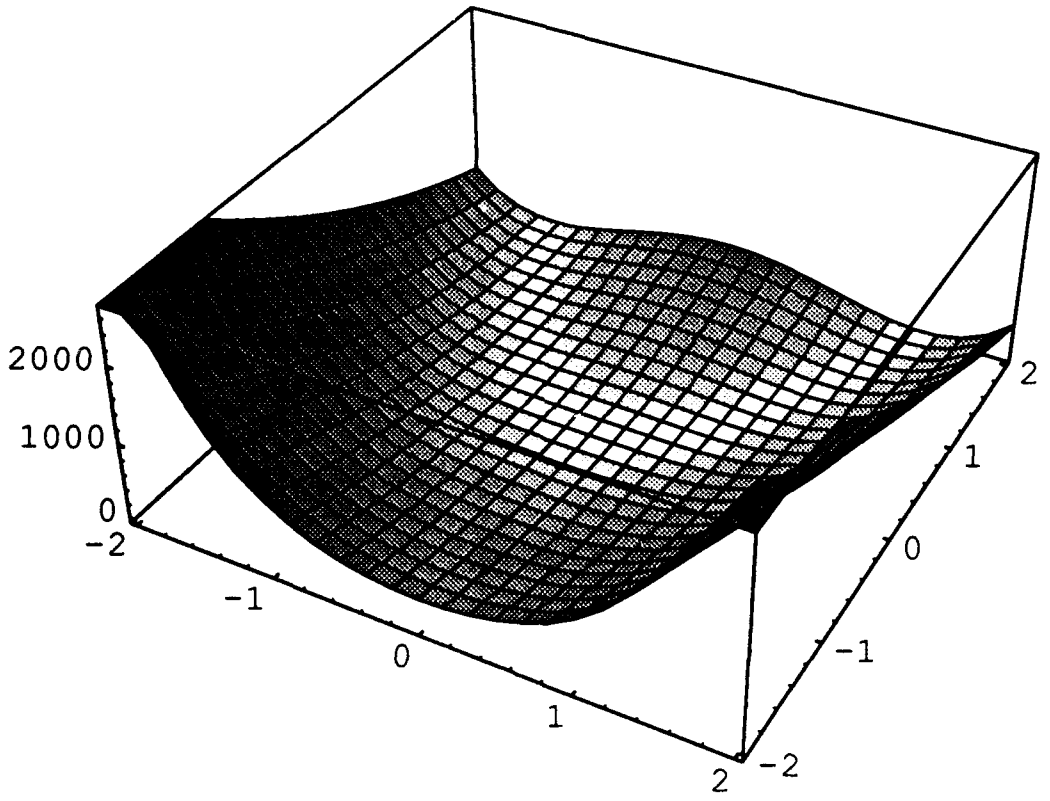


Figure 12: Linear Plot of Rosenbrock's Saddle

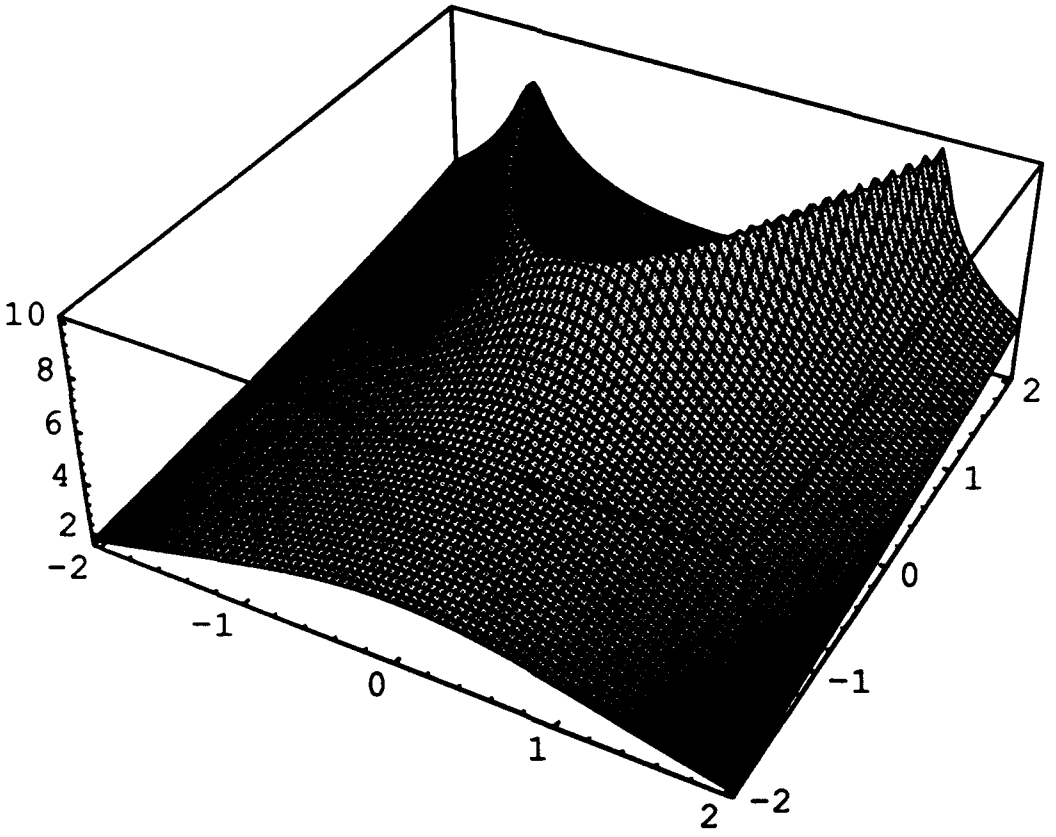


Figure 13: Log Plot of Rosenbrock's Saddle – Front View

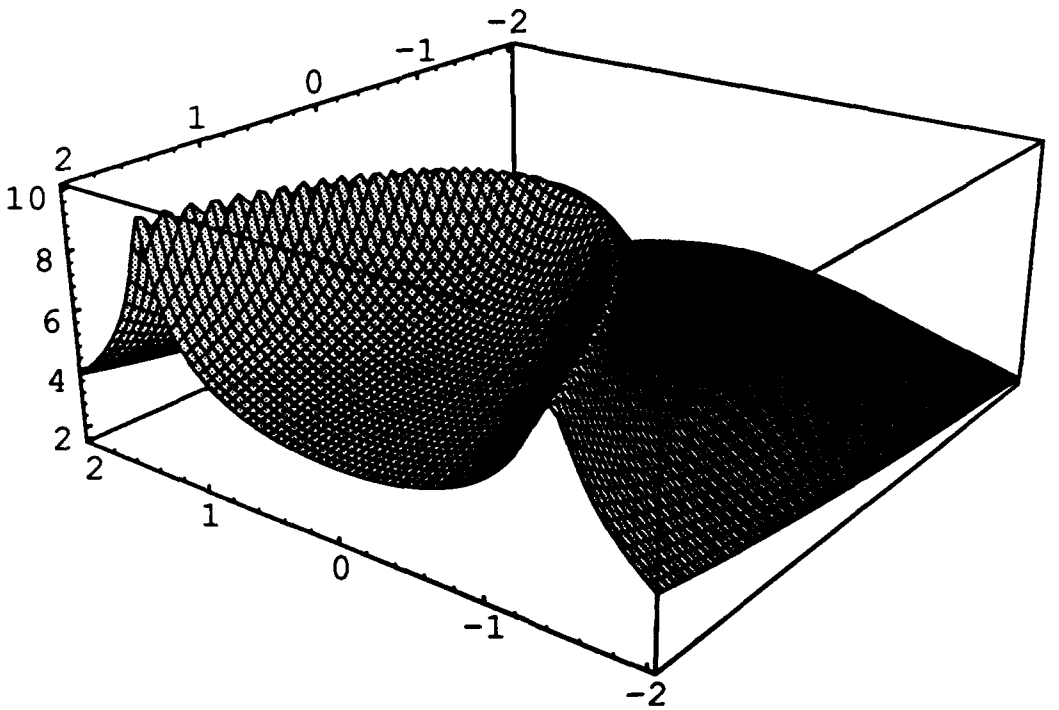


Figure 14: Log Plot of Rosenbrock's Saddle - Rear View

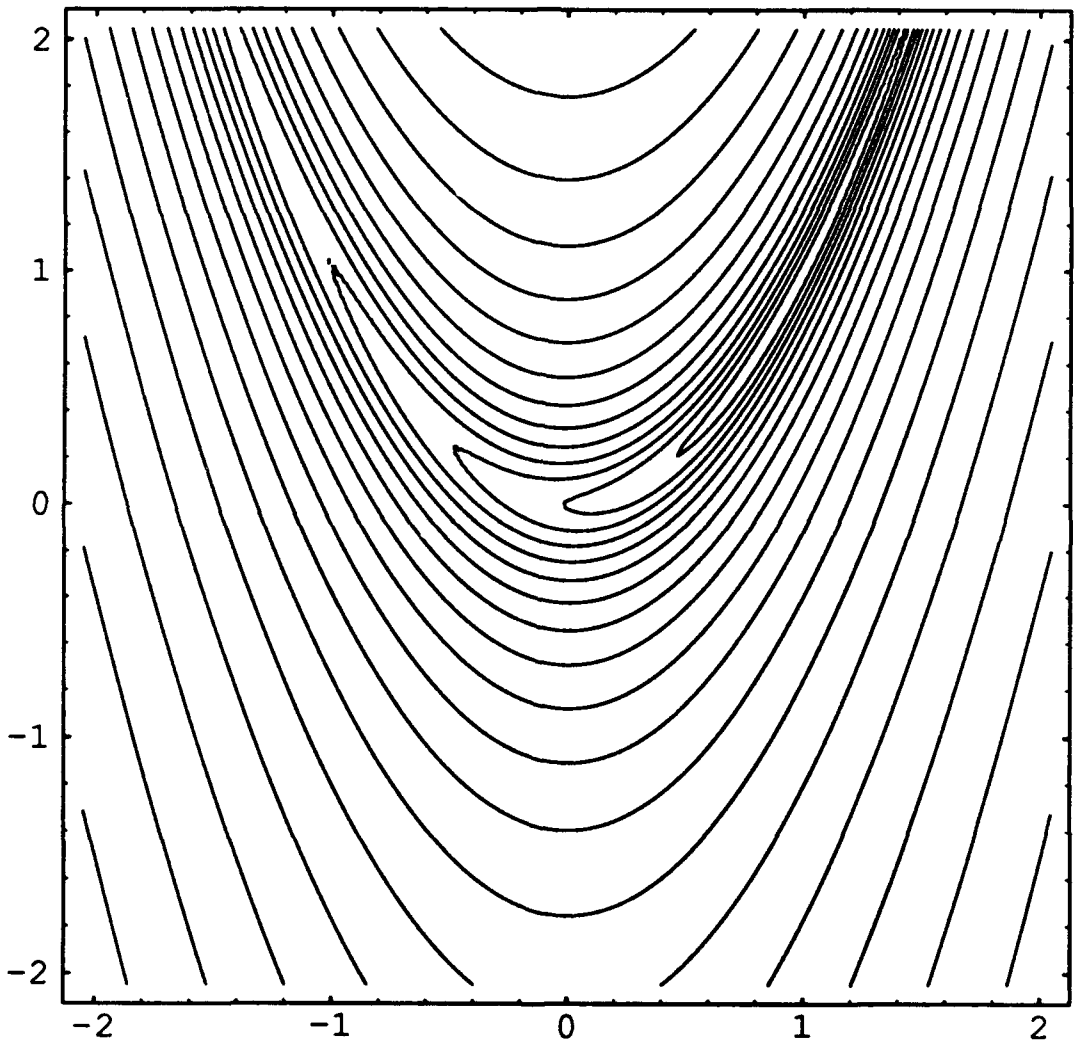


Figure 15: Contour Plot of Rosenbrock's Saddle

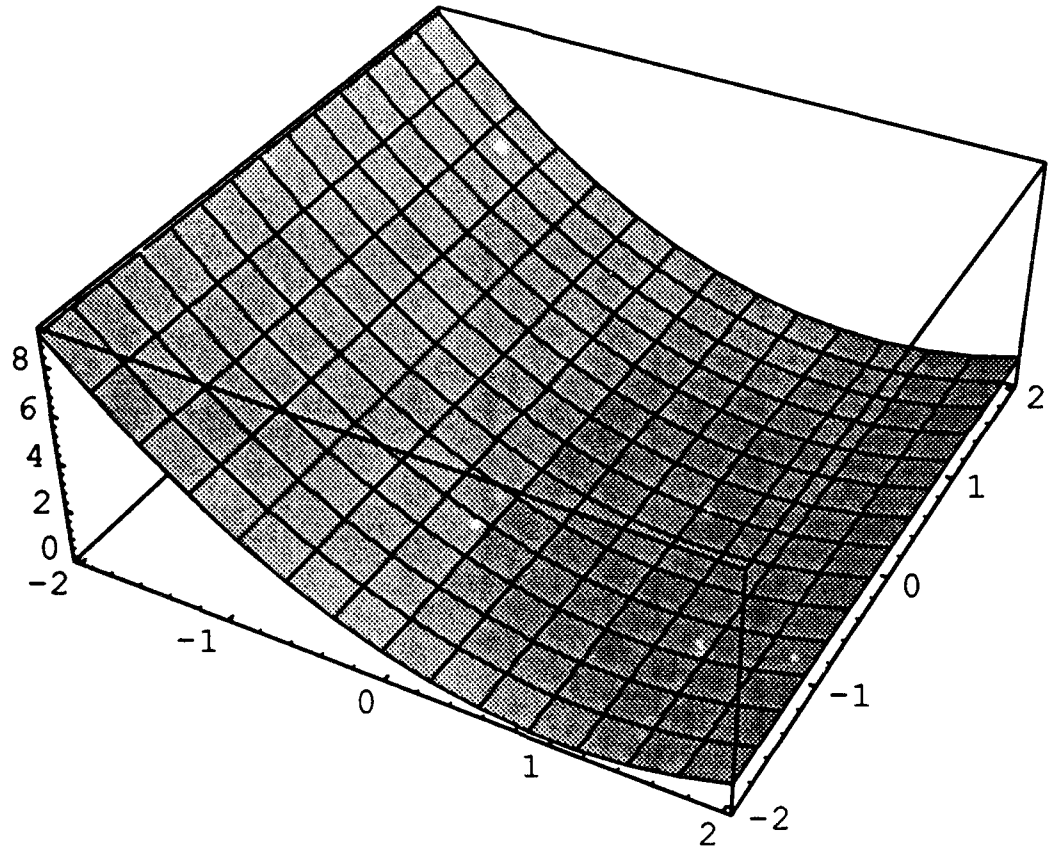


Figure 16: Second Order Term

(crossover, mutation, and inversion) are very disruptive. So rather than "walking the tightrope" of the parabola towards more optimal solution, the search repeatedly climbs and falls off the saddle. As there is no mechanism in a genetic algorithm to prevent redundant search, the genetic algorithm is liable to search only a narrow portion of the search space.

While one can see why the function might be difficult for a genetic algorithm, the function is not GA-hard. GA-hard functions are characterized by an isolated global maxima (4), and such is not the case here. So one suspects that a simple genetic algorithm could be used to find better solutions. The strategies to be used are justified in the next section.

3.3 Preliminary Experimentation and Analysis.

To gain insights into the solution space of Rosenbrock's saddle and the performance and operation of the simple genetic algorithm, the following preliminary experimentation and analysis was performed using an enumerative search program.

3.3.1 Justification. The relatively low dimensionality (2^{24}) of the parameterized search space made it amenable to enumerative search on the Hypercube. The enumerative search provided insights and information regarding the following areas:

1. The cost involved in encoding-decoding the parameters in a To isolate the cost of encoding the parameters as a binary string, two enumerate searches—one which required decoding and one which did not—were carried out on the Hypercube binary string.
2. A benchmark for judging the efficiency of the genetic algorithm.
3. The relative degree of optimality of previous solutions to the problem (Section 3.3.4).

The results and an analysis are contained in the following sections.

3.3.2 Binary Encoding/Decoding is an Expensive Operation. Two enumerative searches were carried out on the Hypercube. The first search encoded the variables as a binary string using the same data structures as in Grefenstette's genetic algorithm. As in the genetic algorithm, the binary string has to be first decoded before being evaluated. In the second search, no such encoding/decoding was required since the variables were represented by decimal digits. Both searches were partitioned among the 8 nodes of the iPSC/2 Hypercube. The run times are shown in Figure 17.

Search	Run Time (sec)
Binary String	2525
Decimal	166

Figure 17. Enumerative Search Run Times

Since the the search space is the same in both cases, the difference in the run times may be attributed to the decoding required with the binary strings. In this respect, the genetic algorithm is handicapped relative to the enumerative search. Just searching a smaller part of the search space is not necessarily sufficient for the genetic algorithm to be faster than the enumerative search. The execution times imply the search space covered by the genetic algorithm must be much smaller.

3.3.3 Enumerative Search has no Redundancy. Another factor supporting the enumerative search is the fact that there is no redundancy. The exhaustive nature of the search is rightly cited as being undesirable. But while each possible solution is considered, it is only considered once. In a genetic algorithm, no mechanism exists to prevent redundancy. A solution produced and evaluated in the first generation may be repeatedly produced and generated in subsequent generations.

The inefficiency associated with the binary representation and redundancy must be countered by the selection mechanism which "guides" the search.

3.3.4 Existence of Local Minima. Pettey's sequential genetic algorithm converged at about 0.0005. All his parallel results were worse (63:158). Knowing only of the existence of the global minima of 0, really does not give a good idea of the effectiveness of the search. Are there any intermediate solutions in the parameterized search space? The more intermediate solutions there are, the worse the genetic algorithm performed. The enumerative search found 165 solutions less than 0.0005.

3.4 Justification of Global Selection on Hypercube.

3.4.1 Theoretical Basis. In a "true" genetic algorithm, selection is based on an individual's fitness in relation to the entire population. The expected number of offspring of an individual A_h is equal to its fitness μ_h divided by the average fitness of all individuals $\mu_h/\overline{\mu}$ (47:94):

$$expected \;\; offspring = \mu_h/\overline{\mu}$$

Since the average fitness is simply the total fitness $\sum_{h=1}^{M} \mu_h$ divided by the total number M of offspring (47:94), the previous equation may be written as follows:

$$expected \;\; offspring = \frac{\mu_h * M}{\sum_{h=1}^{M} \mu_h}$$

$$expected \;\; offspring = \frac{\mu_h * M}{Total \;\; Fitness}$$

Now say a node of a Hypercube has X individuals on it. The expected number of individuals on that node in the next generation is obtained by summing expected number of offspring for each individual on the node:

$$expected \;\; population = \frac{\sum_{h=1}^{X} \mu_h * M}{Total \;\; Fitness}$$

Defining the sum of the individual fitnesses on a node to be the nodal fitness gives the following:

$$expected\ \ population = \frac{Nodal\ \ Fitness * M}{Total\ \ Fitness}$$

Recall each node of the Hypercube is typically assigned M/n offspring and this population size is held constant. Now let us define a term called *population deviation* to be the absolute difference between the expected population size and the constrained population size, M/n:

$$population\ \ deviation = expected\ \ population - M/n$$

3.4.2 Example. Consider a typical Hypercube PGA (parallel genetic algorithm) with a global population size of 200. Each node of the Hypercube is assigned a population size of 50. If after a particular generation the nodal fitnesses are 20, 40, 10, and 30 (global fitness = 100, average fitness = 25), the expected population and expected population error are as shown in Figure 18. So the nodes with above average nodal fitnesses are given fewer offspring than "natural selection"

Node	Fitness	Expected Population	Constrained Population	Deviation
0	20	40	50	-10
1	40	80	50	+30
2	10	20	50	-30
3	30	60	50	+10

Figure 18. Deviations from Expected Population Sizes Using Local Selection

would call for, and the nodes with below average fitnesses too many. Here is a key difference from the sequential genetic algorithm, in which no such "deviations" occur.

Most parallel implementations of a program are functionally equivalent to their sequential counterpart, only faster. A parallel genetic algorithm (PGA), on the other hand, is fundamentally different than a sequential genetic algorithm (SGA). As such, the performance in terms of solution quality are likely to differ. While it has been shown theoretically that a PGA still can search a

search space efficiently (64), the "lessons learned" gathered over the past 15 years in the use of sequential genetic algorithms might no longer apply. That is, if the SGA and PGA are different, and effective parameters have been discovered for use with the SGA, at the very least one would want to re-validate that they are still effective when the code is converted to a PGA. On the other hand, if the SGA and PGA are functionally equivalent, no such re-validation would be required.

This situation is one reason for investigating global selection in this research effort. Global selection would make solutions compete for population slots with solutions on the other nodes rather than just locally. Thus, global selection in a PGA would be functionally equivalent to the selection mechanism in a SGA. Since the SGA outperformed the PGA against Rosenbrock's saddle, the hope is that global selection will improve the PGA's performance against this problem. Having variable population sizes on the nodes may prove beneficial as well. Studies have shown that there is typically an optimal population associated with a genetic algorithm. Since the optimal population size is not generally known, experimentation is required to determine it. Perhaps by allowing population sizes to vary, an optimal population distribution will evolve.

Global selection would not make a PGA identical to a SGA, though. In a SGA, crossover may occur between any two nodes. Allowing crossover between solutions on different processors would require substantial communications. For example, with nodal populations of 100 and a crossover rate of 0.6 on a 8-node Hypercube, on average $(100) * (0.6) * (7/8) = 52$ of the solutions would undergo crossover with solutions on other nodes. Doing this on a Hypercube, with its relatively high communications cost, would likely lead to very high run times, so global crossover will not be considered.

3.4.3 Answering Objections to Implementing Global Selection. Before beginning a full-scale examination of global selection, testing was done to give confidence that the arguments given in its support were justified. Two likely objections and their arguments are presented, along with test results which refuted the objections.

- Objection 1: In practice, the expected population error is zero or close to it.
  - Argument: Each node typically is run with the exact same adaptive plan, and each node has the exact same population size. So each node will produce good and bad solutions with equal probability. As a result, even though the exact same individuals will not be produced on the nodes (due to different random number seeds), the nodal fitnesses remain roughly equal.
  - Disproof of Objection by Example: Figure 19 shows the population changes observed on two of the eight nodes of a Hypercube when global selection was implemented. The starting nodal populations sizes were 300. Population sizes vary rather significantly. With the typical PGA, the population sizes would have been held constant at 300. This would not have been "natural."

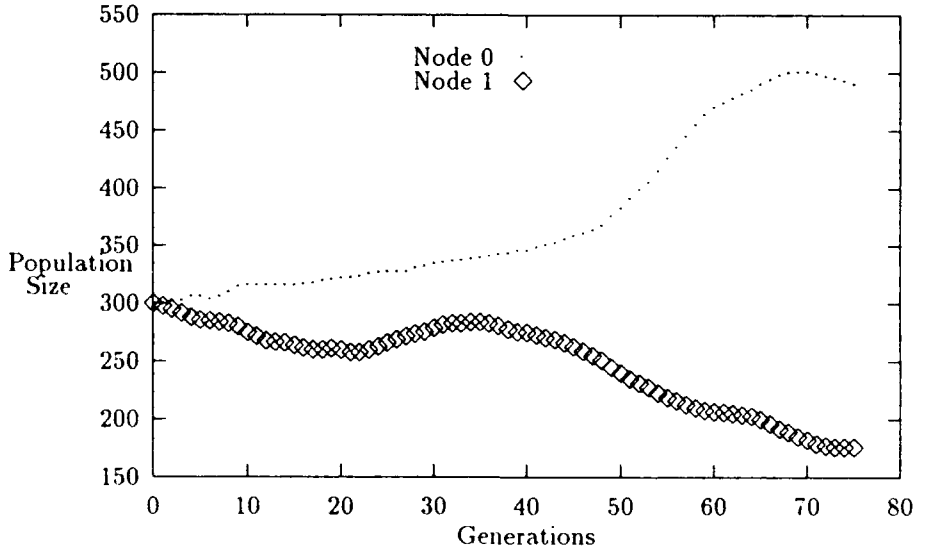


Figure 19. Population Sizes not Constant

- Objection 2: Global selection would lead to "instability." An initial small difference in fitness would quickly result in one node getting the entire population.

- Argument: A difference in fitness, even a small one, would result in one or more nodes getting more individuals in the next generation. Everything else being equal, having "more grist for the mill" would make it likely that the node(s) with more individuals would produce even better solutions during crossover and mutation, resulting in the node being allotted even more individuals. This process would escalate ("domino effect"), and one or two nodes would quickly get most of the population. Due to the tremendous load imbalance, run times would not be competitive with the typical PGA (inherently load balanced).

- Countering of Objection by Example: Figure 20 shows that an initial above average nodal fitness does not necessarily lead to complete domination. Run times are somewhat slower, though, because load imbalance did generally occur. The amount of load imbalance is dependent on the random number seed.

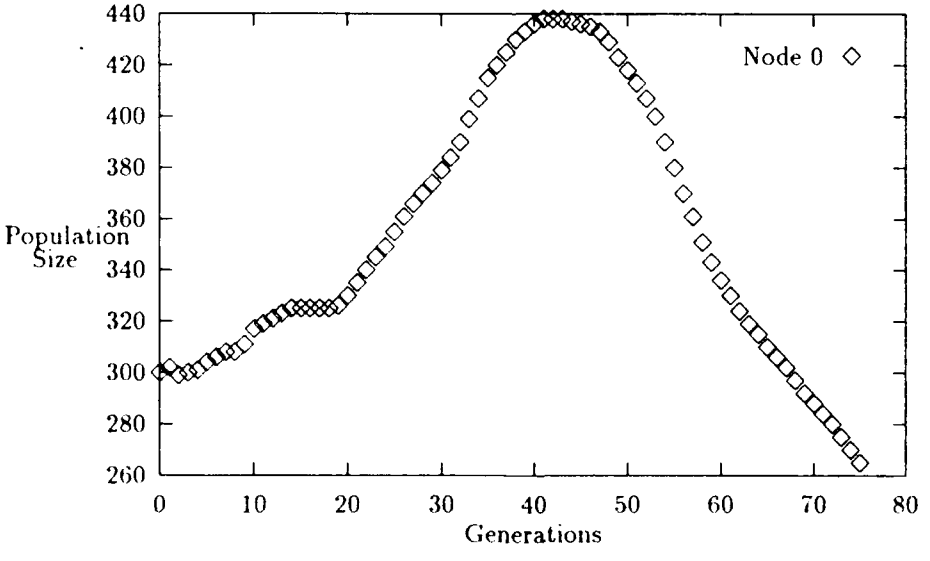


Figure 20. Population Changes are not Monotonic

As the preliminary findings did not show global selection to be feasible, it was decided to implement various global selection strategies in the hope that this would reduce premature convergence.

3.5 Experimental Design.

3.5.1 Code Reuse. The decision to reuse existing code was an easy one. Grefenstette's *Cenesis* genetic algorithm (41) is programmed in C, a language well-suited for use on the Hypercube. Additionally, the modular design of *Genesis* (Figure 21) facilitates modifications. Since Unity is often used in the program design at AFIT, a Unity representation of the *Genesis* was also constructed (Figure 22). Sawyer parallelized this code for a term project (71) using the typical Hypercube decomposition. Sawyer's code became the foundation for the Genetic Algorithm Toolkit and for research into ways for reducing premature convergence.

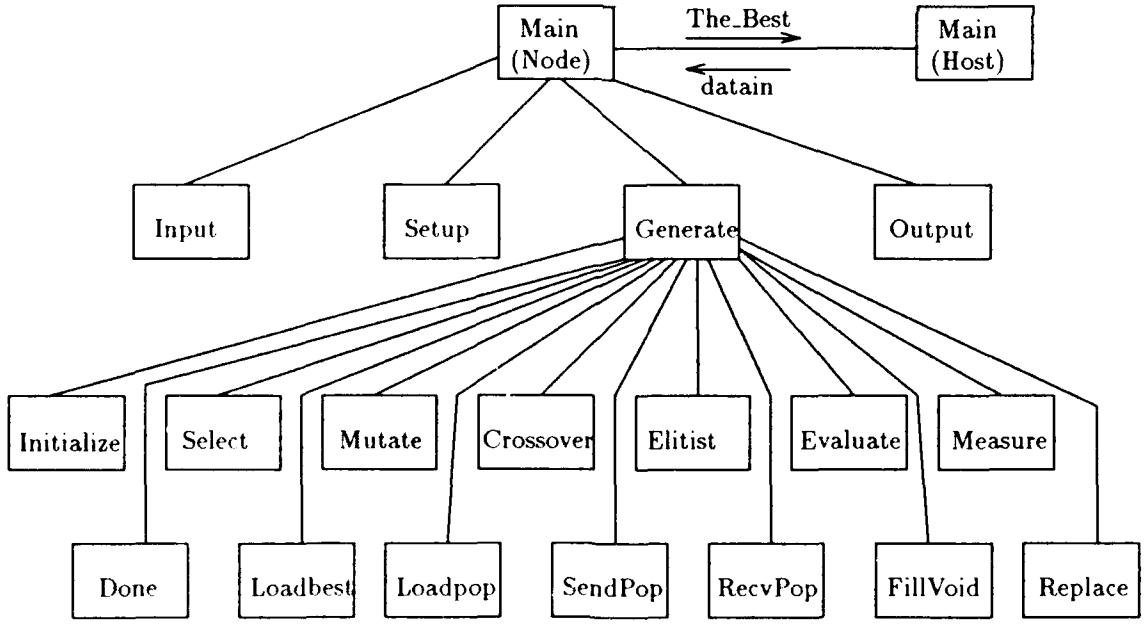


Figure 21. Structure chart of the Parallelized *Genesis Genetic Algorithm*

```
Program Genetic
declare
        the_best : integer;    {the best solution found}
        gen : integer;   {the current generation being considered}
        type allele is 0,1   {a bit in the genetic code}
        type structure is record   {type for a gene (solution)}
        begin
           gene : array[1..MAXSTRUCTSIZE] of allele
             {MAXSTRUCTSIZE is max. length of a gene}
           perf : integer;
             {perf is the value obtained when the genetic code is decoded}
             {and applied to the function being optimized}
        end record;
        type solutions is array[1..POP_SIZE] of structure
        pop : array[1..MAX_GEN] of solutions;
          {A population (pop) has POP_SIZE genes}
        done : array[-1..MAX_GEN-1] of boolean;
          {done is a boolean array used to ensure a population }
          {is not evaluated until after it is created. MAX_GEN is}
          {a constant indicating the number of generation cycles}
          {which will be completed in search of a solution}
initially
        the_best = 0;
        gen = 0;
        pop = initialize(pop);
          {initialize is a function that creates initial values for genes}
        ⟨|| gen  :  1 ≤ gen ≤ MAX_GEN  ::  done[gen] = false⟩
          {No generation is completed}
        done[-1] = TRUE;   {to prevent boundary error on 1st generation}
assign
        ⟨|| gen : 0 ≤ gen < MAX_GEN  ::  pop[gen], gen, the_best, done[gen] :=
          crossover(mutate(select(pop[gen]))), gen + 1, evaluate(pop[gen]), TRUE
                          if done[gen-1] = TRUE ⟩
          {select-reproduce, mutate, and crossover (functions)}
          {must be done in this order and only if this process has}
          {been completed on the previous generation}
          {evaluate compares the_best with all solutions}
          {in the current population and should a better}
          {solution exist, it replaces the_best with it}
    ||
        the_best := evaluate(pop[MAX_GEN]) if done[MAX_GEN-1]
          {a boundary condition—notice quantified assignment}
          {does not evaluate last population}
end {Genetic}
```

Figure 22. Top-Level Unity Design of a Genetic Algorithm

3.5.2 Code Evaluation. Prior to conducting the experiment, the code was tested for errors. Since Grefenstette's code has been in the public domain for quite some time, the testing concentrated on the changes Sawyer made to parallelize the code (71), principally those changers dealing with parallel random number generation and communications.

1. Parallel Random Number Generation. The stochastic features of a genetic algorithm are implemented with a pseudo-random number generator (see discussion in Appendix A). The random number generator was examined in detail due to its importance in a stochastic algorithm like a genetic algorithm. Experimentation with input seeds showed that for small seed values, the nodes returned identical (and generally poor) solutions. The method of assigning random number seeds to a node was found to allow for identical seeds, resulting in redundant search. The details on how this was corrected are contained in Appendix A. Additionally, a chi-square test for randomness was performed, which the random number generator passed. The details of this test are also contained in Appendix A.

2. Communications. The fitnesses of communicated solutions were being somehow set to 0. This resulted in disproportionate reproduction and the appearance of finding the global optimum on every run. A minor change corrected this problem.

3.5.3 Implementation of Global Selection. Selection in a genetic algorithm involves allotting population slots based on "survival of the fittest." Crossover and mutation generally result in significant changes in the population. Without a selection mechanism, the population would consist of a random mixture of good and bad solutions. Selection provides direction to the search by creating a new population in which the number of slots allocated to a solution is proportionate to its fitness relative to other population members. So to perform global selection, global fitness information was required. The steps involved were

1. Calculate the total fitness of solutions. The command *gdsum* (global, double-precision floating point sum) was found to do this quite efficiently. Benchmark tests showed a *gdsum* to take 1.56 ms, so even when performed each generation, calculating total fitness was a small percentage of the execution time (see execution times in the Chapter IV).

2. Determine the worst fitness. This being a functional minimization problem, the smaller the value returned by the function, the fitter the solution. One way of converting a low functional value to a high fitness value is by subtracting the functional value from a large number. In his *Genesis* program, Grefenstette subtracts the functional value of a solution from the functional value of the worst solution. In conjunction with performing global selection, the global worst solution had to be found. To do this, each node made a call to the Hypercube function *gdhigh* with its local worst solution as a parameter. The global worst solution value was returned. Since global commands require the nodes to be temporarily synchronized, the *gdhigh* command was placed immediately after the *gdsum* command, so the nodes would not have to be re-synchronized.

3. Calculate expected values. The average fitness of a solution $\mu_h/\overline{\mu}$ was calculated by dividing the total fitness by the total population size. The expected values of each solution E_h was then calculated as follows:

$$E_h = \frac{Worst - \mu}{Worst - \mu_h/\overline{\mu}}$$

4. Probabilistically create a new population based on expected values. The most intuitive way of implementing this is with a roulette wheel. Each solution is given a number of slots on the wheel in proportion to its fitness. The wheel is then spun and the solution on which it stops is added to the new population. This process is repeated until all population slots are filled. "Spinning wheel" selection, also known as "Stochastic Sampling with Replacement (SSR)" (1:15), was the method of choice of Goldberg in his work (34). While SSR has the positive quality of not being biased (that is, selection probability = expected value), it is relatively

inefficient, having a time complexity of $O(N^2)$ typically (N=Population Size) or O(NlogN) if a B-tree is used (1:14-15). To improve efficiency, Baker developed a new selection algorithm called "Stochastic Universal Sampling which is bias-free and has a time complexity of O(N) (1:16-17,19). A C code fragment is as follows:

```
ptr = Rand();     /* Wheel pointer set equal to random number [0,1] */
for (sum=i=0; i < N; i++)   /* N is the size of the population */
    /* increment ptr until > expected value E */
    for (sum += E[i]; sum > ptr; ptr++)
        Select(i);      /* include individual in new population */
```

Grefensttete used SUS in his *Genesis* code. A disadvantage of SUS for a parallel computer is that it is "strictly sequential" (1:16). To perform Baker's algorithm on a population distributed among the nodes of a Hypercube, one node would have to generate the initial random value of the pointer, calculate the expected values of its population values, then pass on the current values of the *ptr* and *sum* values to the next node. This node would calculate its expected values, then again pass on *ptr* and *sum* to the next node. This process would continue until all nodes have performed selection. The time complexity O(N) and communications complexity is O(N). Strictly speaking, Baker's claim that SUS is nonparallelizable is correct.

Yet if a little "sloppiness" is allowed, Baker's algorithm can be parallelized. When a node completes selection, the *ptr* variable would be greater than *sum*, but the difference would be less than one. Rathe. than sending values of *ptr* and *sum* in sequence, each node could set *ptr* equal to a random number between 0 and 1, and sum to 0. Again, *ptr* would be greater than *sum* with a difference less than 1. Of course the values would not be the same as in the "strict" version. Since no communications are used, the time complexity is $O(A)$, where A is the largest population size. Using the roulette-wheel analogy, if Baker's original algorithm is

like spinning one large wheel, the parallel version is similar to spinning one smaller wheel on each processor (Figure 23).

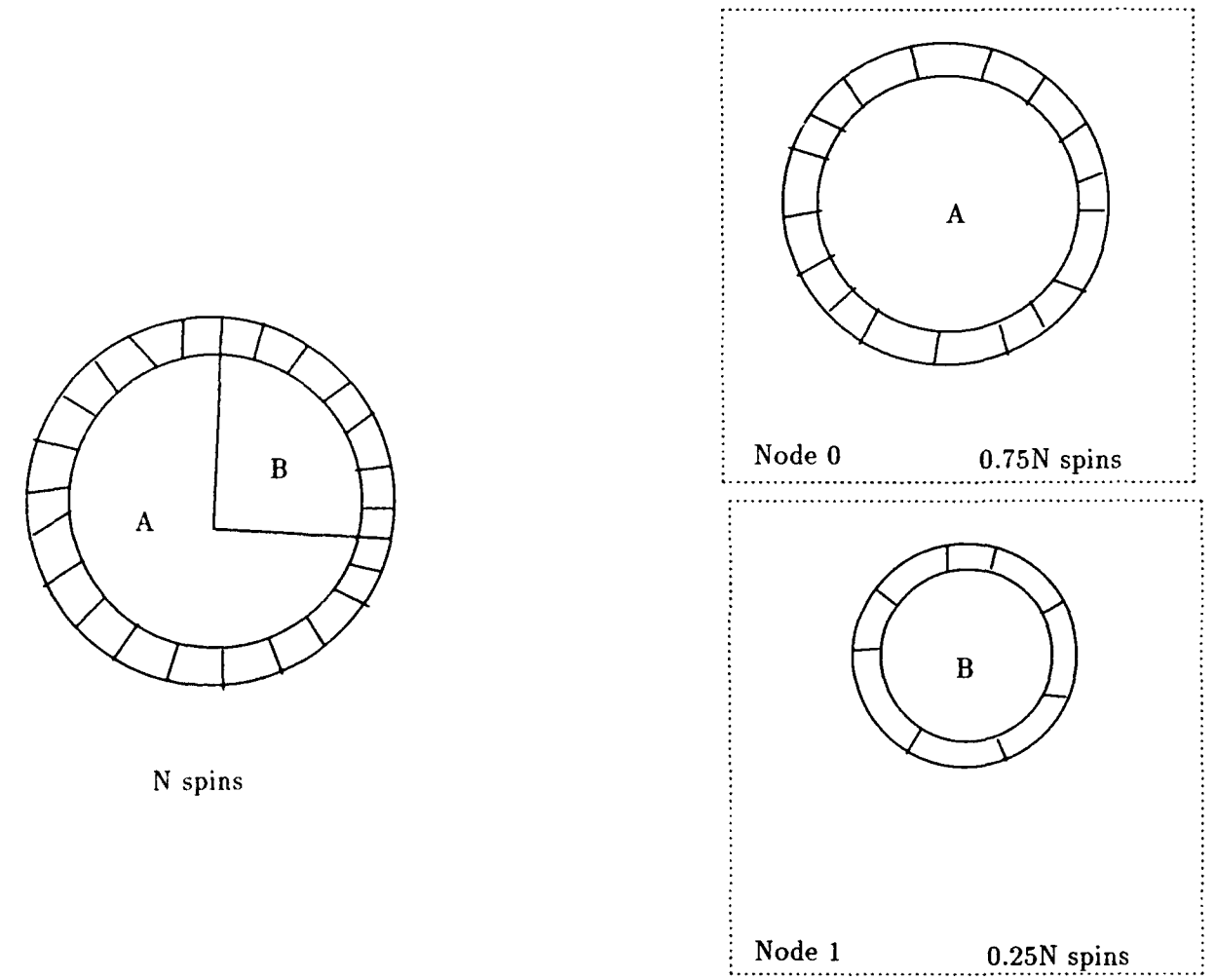


Figure 23. Parallelizing Baker's Algorithm

To allow global selection, the data structures for the populations had to be modified to accommodate variable populations. For simplicity, large fixed arrays with a counter for the current population size were used. The intent was that if global selection was found effective, the population data structure could be converted a more space efficient linked-list structure.

3.5.4 Parameter Settings/Selection Strategies.

1. Repetitions. Trial runs showed that solution quality is quite dependent on the random number seed (Figure). For the variable-population schemes, run times were also quite dependent on the seed. To get maximum-likelihood estimates, it was decided to take averages over 40 runs.

2. Standard Parameter Settings. A crossover rate of 0.6 and a mutation rate of 0.001 are the most frequently mentioned settings in the literature, probably because these generally return good results. While increasing the mutation rate might be tried to reduce premature convergence, the disruptive nature of mutation would seem at odds with staying close to the narrow parabola of Rosenbrock's saddle. So it was decided not to vary the crossover and mutation rates, but rather to use the settings mentioned above. Instead, it was decided to vary the population size. Goldberg suggests using large populations with parallel genetic algorithms, and the population sizes used in the past were quite small. A large population would seem more resistant to premature convergence since it would take longer for a locally optimal solution to dominate the population. Thus it was decided to vary the nodal population size to see whether superior solutions to those reported by Pettey could be obtained.

3. Total Generations. Since the optimal value of function f2 is known, the genetic algorithm could be stopped as soon as the optimal solution is found. It was decided not to do this. Knowledge of the optimal result is not normally available, otherwise the search would not be conducted. Using this information would hide one of the genetic algorithm's biggest weaknesses—lack of an adequate stopping criteria. Following the example of most implementations in the literature, it was decided to terminate the genetic algorithm after a fixed number of generations. All of Pettey's runs had converged long before 100 generations (63:147). Since the strategies used in this research were expected to delay convergence, it was felt if the genetic algorithms were terminated after 100 generations, productive search might still be ongoing. To avoid premature termination, the genetic algorithms were terminated after 200 generations.

4. Communications. To maximize the effect of communications, for good or bad, it was decided to send the best solution on a node to all nodes rather than the just the nearest neighbors. In benchmark tests, the global communication (using the *gsend* command) was found to be quite efficient, taking an average of only 1.56 ms on an 8-node Hypercube. In fact, the global communication of solutions was found to take no more time than exchanging solutions among the 3 nearest neighbors (using 3 consecutive *csend* commands and a preliminary calculation to determine nearest neighbors). Evidently, Intel has employed an efficient tree method to implement global send, requiring just $\log_2 8 = 3$ steps to complete. To clearly demarcate the effect of a communication in the strategies in which solutions were exchanged, it was decided the epoch length should be 5 generations. Had solutions been exchanged every generation, there would be no basis of comparison.

5. Strategies. The five strategies used were

   (a) Local with Sharing (LS). Selection is local and the best solutions are shared as above. The resulting genetic algorithm is similar to the Hypercube implementations reported in the literature (63) (73).

   (b) Local with No sharing (LN). Selection is local and solutions are not shared. So the nodal genetic algorithms are completely independent. The fact that this strategy is equivalent to 8 separate sequential genetic algorithms *guarantees* improvement over Pettey's results, in which 1 sequential genetic algorithm outperformed the parallel implementation.

   (c) Global with Sharing (GS). Selection is global and the best solutions are shared as in LS.

   (d) Global with No sharing (GN). Selection is global and no communication of solutions occurs.

   (e) Global, no sharing, Parallel select (GP). Selection is global, no communication of solution occurs, and Baker's selection algorithm is parallelized.

3.5.5 Data Gathering. In choosing what data to gather, an attempt was made to focus on the information most pertinent to an optimization problem. Two common "criteria of goodness" often used to gauge genetic algorithm performance are on-line performance and off-line performance (34:107,110). On-line performance is a running average of the fitness of *all* solutions. Off-line performance is the average of the best solutions from each generation. Neither really seemed to focus on the most important solution—the overall best solution. In an optimization problem, generally one is not interested in several good solutions, but the one best solution.

To reflect this goal, rather than gathering data on on-line and off-line performance, the focus was placed on the overall best solution. During each generation, each node saved its best solution. No averaging was done as with off-line performance (this would "mask" the overall best solution with the inferior "best" solutions from early generations). At the completion of the run, the solutions from the nodes were compared, and the global best solution from each generation was retained. In this way, the evolution of the overall best solution through the 200 generations was charted.

Two additional measures were made. First, the run time of the genetic algorithm through 200 generation on the node was recorded. Notice this excludes the strategy-independent time it takes to load the nodes and to communicate the solutions/data to the host. Comparing the run times would show the relative efficiencies of the different strategies. Additionally, each node recorded the generation in which its best solution was found. At the conclusion of the run, the generation in which the overall best solution was found was retained.

Trial runs showed that solution quality is quite dependent on the random number seed. For the variable-population schemes, run times were also quite dependent on the seed. To get maximum-likelihood estimates, averages over 40 runs were taken.

IV. Results of Premature Convergence Reduction Strategies

4.1 Introduction.

This chapter summarizes the results of the application of the local and global selection strategies described in the previous chapter. Comparisons are made between the selection strategies in terms of solution quality, execution time, premature convergence, and the likelihood of returning the optimal solution. Additionally, the Goldberg's theoretically predictions of optimal population size are compared to the experimental results.

4.2 Data Compression and Interpretation.

Figures 72 - 87 show the evolution of the best solution averaged over 40 runs for various population sizes. Figures 88 - 103 show the average, variance, and standard deviation of the best solution, the generation in which the best solution is found, and the run time for the 200 generations.

The following observations can be made:

- Rosenbrock's Saddle is not an "intractable" problem for a parallel genetic algorithm. The LS strategy found the optimal solution by generation 180 on all 40 runs at a population size of 3200 (Figures 87 and 103), and in at least 36 of the runs in population above 1920. Additionally, the optimal was found in a high percentage of the runs for all strategies at the higher population sizes.
- There is some agreement with Goldberg's theoretical predictions as to an optimal population size. Goldberg has developed a theory concerning the optimal population size of a binary coded genetic algorithm (30). Developed originally for sequential genetic algorithms, the theory predicts solutions will improve up to the optimal population size, then solution quality will deteriorate or show no improvement. For a 24-bit string used with Rosenbrock's saddle (function f2), the predicted optimal population size is 51 (63:155-156). While Pettey's results

seemed to corroborate the theory for other test functions, Rosenbrock's saddle was the exception (63:156). The results here seem to indicate that the optimal population size for f2 is much higher.

Goldberg has since revised his theory to include predictions for the optimal population size of parallel genetic algorithms (35). For the degree of parallelization and string length used in this implementation, Goldberg predicts an optimal population size of approximately 2000 (35:75). The optimal population sizes for two of the three strategies that do not share solutions (LN and GN) as well as and GS show good agreement with Goldberg's theoretical predictions. The optimal population sizes for the various strategies in terms of the average best solution are shown in Figure 24. Figure 25 shows the optimal population sizes for the various strategies in terms of the propensity in finding the optimal solution. Regardless of whether local or

Strategy	LS	LN	GS	GN	PN
Average	0.0	$3.01 * 10^{-7}$	$3.65 * 10^{-6}$	$4.75 * 10^{-7}$	$4.00 * 10^{-7}$
Pop. Size	3200	1920	2880	3200	1920

Figure 24. Optimal Population Sizes in Terms of Solution Average.

Strategy	LS	LN	GS	GN	PN
Best = Optimal	40	31	39	30	29
Pop. Size	3200	1920	2560	3200	1920

Figure 25. Optimal Population Sizes in Terms of Finding Global Best.

global selection is used, sharing of the best solution seems to increase the likelihood of finding the optimal solutions. The GS strategy, while generally the poorest in terms of the average best solution, is second only to LN in returning percentage finding the optimal. Evidently, a few poor solutions adversely affected the average.

- Sharing of best solutions leads to premature convergence at low population sizes. As shown in the Figures, the strategies that use communication return generally poorer solutions than

strategies that do not. There is less useful search over the 200 generations, as indicated by the fact that LS and GS find their best solution earlier than the strategies that do not communicate. This would be welcome if the solution was as good as or better than the solutions from the strategies that do not communicate, but this is not the case.

- Sharing of the best solutions enhances finding the optimal solution at larger population sizes. Even though the average of the best solutions returned by the different strategies may be similar, the strategies that communicate solutions are more likely to find the optimal. In fact, strategy GS, while generally the poorest in terms of average, is second only to LS in the likelihood of finding the optimal. The average of the best solutions returned by GS is tainted by a few inferior solutions.

- Comparing the global select strategies, we see that the strategy that communicates best solutions (GS) is actually faster than the strategy that does not (GN). While this may seem counter-intuitive given that communication takes time, the effect of the communication is a better load balance. When the nodes do not share solutions, should one node find a relatively good solution, this quickly results in a higher nodal fitness and consequently a greater nodal population size. While this increases the likelihood of finding even better solutions on the node with the larger population, the nodes with smaller populations have a lesser likelihood. As a result, the imbalance tends to perpetuate and even increase. One the other hand, should the node finding a good solution share it with other nodes, the chance of a fitness and population disparity is lessened. Intuitively, it would seem that the more often sharing occurs, the less the load imbalance, and vice versa. Notice that the nodes exchange the same number of solutions which other. That is, the load balancing achieved by communication is indirect rather than direct.

- In 13 of the 16 population sizes used, a strategy using global select outperformed the strategies using local select in terms of the average best solution. The exceptions were at a population

size of 80, 960, and 3200. Thus, the global strategies tended to be more "robust." A user not knowing the optimal population size would have been been better off using a global population strategy.

- In terms of run time, the strategies using local select were vastly superior. The load imbalance associated with the global select, while leading to a favorable population distribution, does adversely affect run time. Looked at another way, the local select strategies can operate on larger populations in the same amount of time than the global selection strategies on smaller populations.
- This leads to the following question: Given the same amount of run time, which strategy will produce the best solution on average? The results in Figure 26 show that under this criteria the strategy LN generally returns the best solutions.

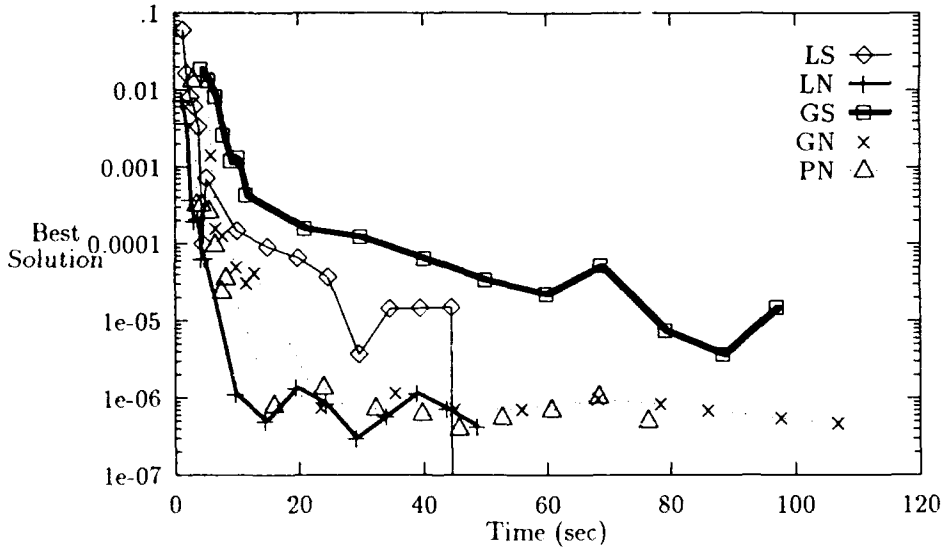


Figure 26. Best Solutions versus Run Time

As indicated by the high standard deviations in the best solutions, solution quality was very dependent on the random number seed seed. For example, Figure 27 shows the best solutions versus the random number seeds for the 40 runs of strategy PN at a population size of 960.

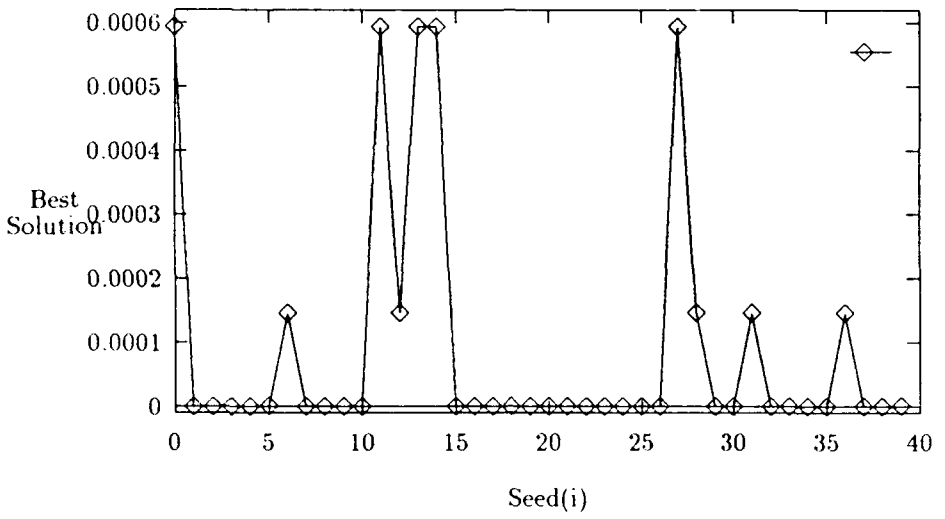


Figure 27. Solution Quality is Dependent on Random Number Seed.

A status of the static analysis of the parallelized *Genesis* code suggests that the execution time of the program should vary linearly with the size of the population, regardless of the selection strategy used. A plot of the experimental data Figure 28 suggests that the linear relationship holds in practice. The method of least square was applied to find the straight lines that best fit the data. The results of the least squares calculations are shown in Figure 29, with t the execution time in seconds and P the total population size. To test the validity of the assumed linear relationship, the correlation coefficient was calculated. The correlation coefficient r, also known as the index of association, is a measure of the strength of the linear relationship between two variables (51:75). Values of r so close to 1 (Figure 29) indicate a very strong positive association between the execution time and population size (51:72). Thus, the substantial improvements in solution quality came at the expense of just linear O(P) increases in execution time.

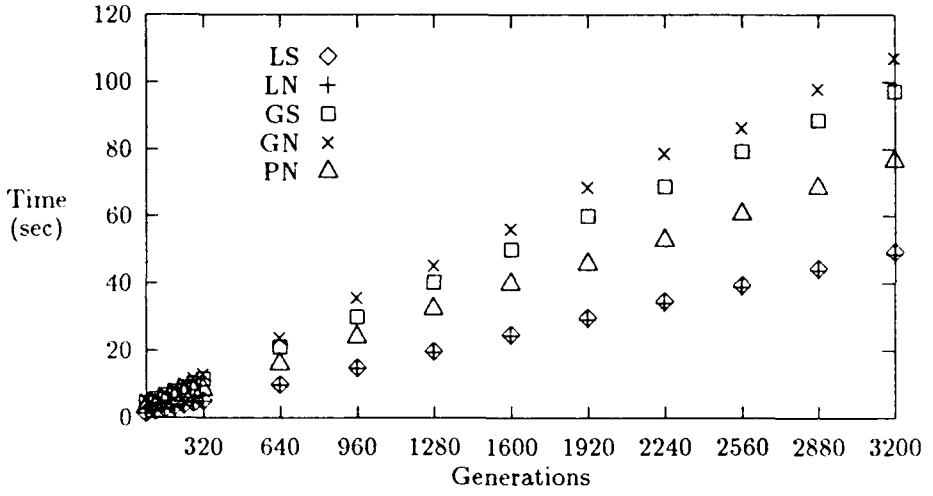


Figure 28. Execution Time Varies Linearly with Population Size.

Strategy	Least Squares Equation	r	r^2
LS	$t = 0.015383P + 0.165297$	1.0000	1.0000
LN	$t = 0.015218P + 0.045728$	1.0000	1.0000
GS	$t = 0.029985P + 1.815469$	0.9999	0.9999
GN	$t = 0.033180P + 2.545763$	0.9996	0.9992
PN	$t = 0.023491P + 1.096119$	0.9997	0.9995

Figure 29. Least Squares Regression Equations.

4.3 Why the Results were not Generalized.

It would be desirable to extend the results observed in this research to all applications of genetic algorithms, but it was thought impossible to do so. Figure 30 shows that an adaptive plan, like the ones used in this research, belongs to the power set of genetic operators. The operators may be applied numerous ways to form subclasses of non-deterministic genetic algorithms (NGA). The instantiation of the non-deterministic algorithms with parameters such as the mutation rate, crossover rate, and population size creates a genetic program ready for execution on a computer. Each parameter may be varied throughout its range, each variation creating a new genetic algorithm (GA). By varying only the selection strategy and population size, the behavior of only a very tiny subset of the universe of genetic programs was observed. Additionally, the genetic programs were applied to only one solution space in the universe of solution space. So while it may be conjectured that, say, increasing population size always reduces premature convergence, this could not be stated definitively.

4.4 Summary.

This chapter presented experimental results obtained when genetic algorithms using selection strategies LS, LN, GS, GN, and PN. The presentation focused on the two most important criteria for a semi-optimal search strategy–solution quality and execution. Additionally, an examination of the experimental optimal populations sizes revealed fairly good agreement with Goldberg's theoretical predictions.

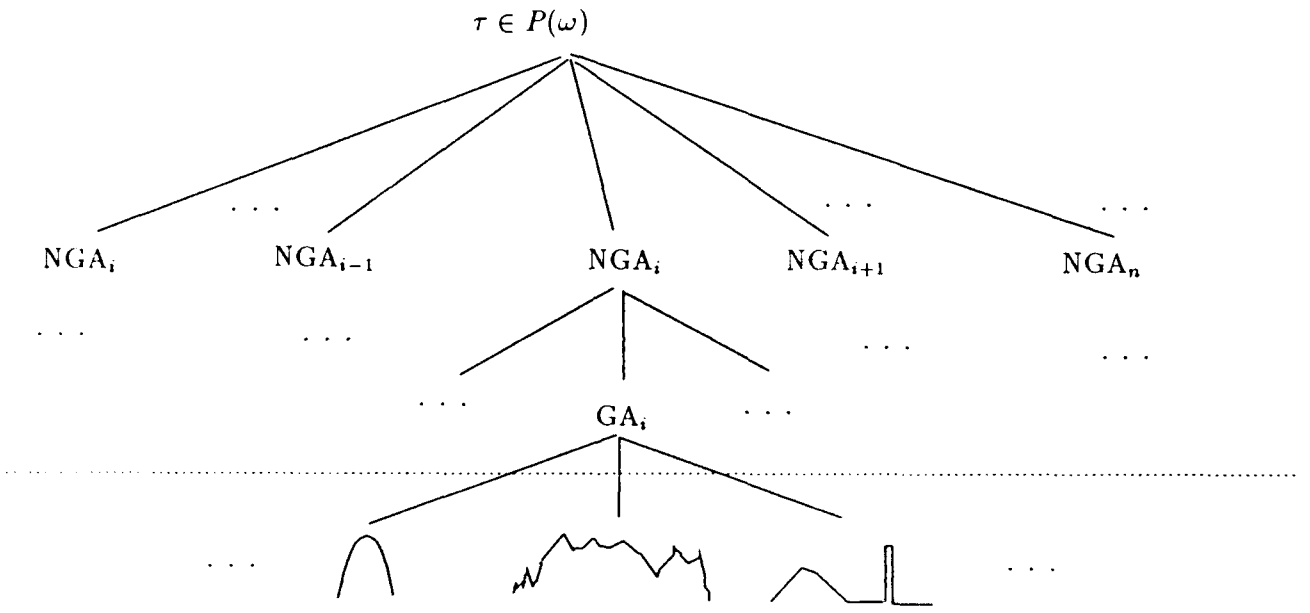


Figure 30. Genetic Program Universe.

V. Methodology & Design - Messy Genetic Algorithm.

5.1 Introduction.

To ensure the description of the messy genetic algorithm as reported in the literature (36) (37) (38) was being correctly interpreted, a design and implementation of a sequential messy genetic algorithm preceded the parallel design and implementation. This chapter describes the sequential design and implementation, while the next chapter does the same for the parallel implementation. The design and implementation proceeded in increasing levels of detail, and such a methodology is reflected in the layout of this chapter. Section 5.2 discusses the general requirements of a messy genetic algorithm. In Section 5.3, the high level design, in which the problem is decomposed into the required objects and operations, is presented. The design is further refined in the section on the low level design (Section 5.4), in which time and space tradeoffs among various data structures and their associated algorithms are discussed. Section 5.5 presents a structure chart and algorithms which served as the blueprints for the coding of the messy genetic algorithm in C. Finally, Section 5.6 discusses the problem chosen to validate the messy genetic algorithm.

5.2 Problem Discussion.

The messy genetic algorithm is a general problem-solving tool which returns a semi-optimal solution. The user must tailor it to the specific problem by developing a means of encoding solutions to the problem using symbols (genic alphabet). The user must also provide an evaluation function which assigns a "fitness" to a string based on well the strings solves the problem. Once the parameter associated with encoding the solution are specified, the genetic algorithm performs operations intended to construct a "semi-optimal" solutions to the problem.

A more formal specification of a program than that given above would be desirable. This was not done because of the inability to specify a postcondition detailed enough to describe the

effect of a messy genetic algorithm. Section 5.7 discusses the reasons for the lack of a satisfactory postcondition.

5.3 High Level Design.

Goldberg and his associates give a high-level description of messy genetic algorithms in their three papers on this subject (36:493-530) (37:415-444) (38:24-30). Based on the information provided in these papers, the following objects and operations are necessary:

5.3.1 Required Objects.

1. *String.* A string is a candidate solution encoded in a form similar to a chromosome. This is done so the string will be amenable to the genetic operations of cut and splice. A chromosome consists of genes have a value (allele) and a position (locus). Similarly, a string in a messy genetic algorithm consists of a list of pairs of numbers, one number specifying a value from the genic alphabet and the other a position in the string. Goldberg represented the string 111 as ((1 1) (2 1) (3 1)), the first number in the pair being the position, the second the value (37:417). A key difference from the simple genetic algorithm is the string may be underspecified or overspecified. That is, the string may contain too few genes to completely solve the problem (partial solution), or the string may contain redundant and/or conflicting genes. Another difference is that the length of the string is variable. The cut operation shortens the strings and the splice operation lengthens strings.

2. *Population.* A large number of strings exist concurrently in a messy genetic operation. During the primordial phase, the number of strings is decreased at regular intervals. To facilitate an operation such as this, a higher level object which encompasses all the strings is desirable. The population is a variable-length list containing all the strings in the genetic algorithm.

3. *Competitive Template.* The competitive template is a "locally optimal" string used to assign a fitness to under-specified strings. The competitive template is complete and non-redundant solution to the problem. That is, the competitive template is neither under-specified or over-specified. Once generated, the competitive template is constant in length and value.

5.3.2 Required Operations. The required operations are organized into the three main steps of a messy genetic algorithm:

1. *Initialize Population.* Based on the user-inputs of the genic alphabet, the string length, and the building block size, the population is initialized by generating all possible strings of length building_block_size. These strings are known as building blocks.
2. *Enrich Population.* The purpose of this operation is to produce a high concentration of good-quality building blocks. Population enrichment occurs during the primordial phase of the messy genetic operation. To achieve the goal of an enriched population, the following sub-operations are required:
   (a) *Selectively Reproduce Population.* This involves creating a new population from the current population, with strings that are fitter getting relatively more population slots in the new generation. Goldberg suggests using tournament selection, a form of selection which seems to inhibit super individuals from dominating a population (36:504-505). Since the need to inhibit super strings is not an explicit requirement, the form of selection to be used remains a design decision to be determined in low level design.
   (b) *Reduce Population Size.* Since the initial population created by generating all combinations of strings of length building_block_size can be very large, the population is halved at regular intervals (36:509). There is nothing special about a 50 per cent reduction, but the developers of the messy genetic algorithm got good results using this setting (36:512-515). Additionally, they performed the reduction every other generation.

3. Generate Solution. Once a population of high-quality building blocks has been created, they are then used to create solutions to the problem. This is known as the juxtapositional phase of the genetic algorithm. The following operations are required:

   (a) *Cut.* A pair of strings is chosen at random. A point is chosen at a random location along each of the strings. Each string is then cut with probability, *cut_probability*. A cut produces two shorter strings from the original string. Since the cut operation is probabilistic, neither strings may be cut, one string may be cut, or both strings may be cut.

   (b) *Splice.* The splice operation is then performed on the strings resulting from the cut operation. A splice is the concatenation of two strings to produce a longer string. Each pair of strings is considered in turn, a splice being performed with a probability, *splice_probability*.

   (c) *Selection.* Once the entire population has undergone cut and splice, the population is reproduced using a selection operator as in the primordial phase.

   (d) *Test for Termination.* While Goldberg does not mention a criteria for ending the program, termination conditions must be defined and tested for. Possible termination criteria might be a limit on the number of cut-splice-reproduce cycles (generations) or a minimum requirement on solution quality.

The data flow diagrams in Figures 31 - 34 give a pictorial representation of the high-level design. The data dictionary is shown in Figure 35.

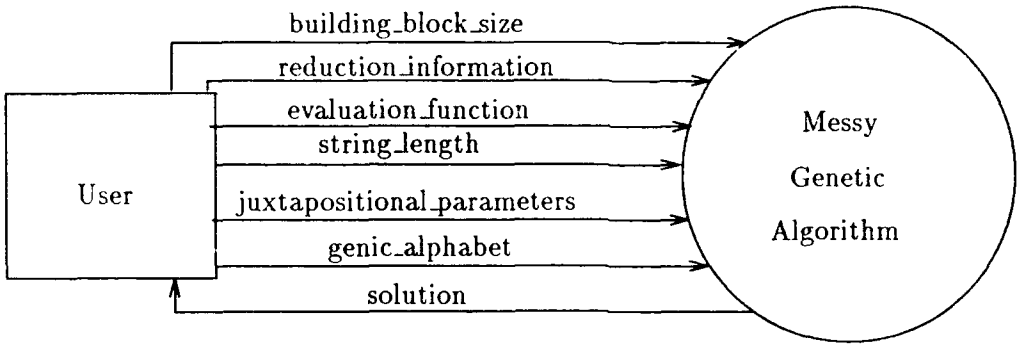


Figure 31. Context Diagram for a Messy Genetic Algorithm

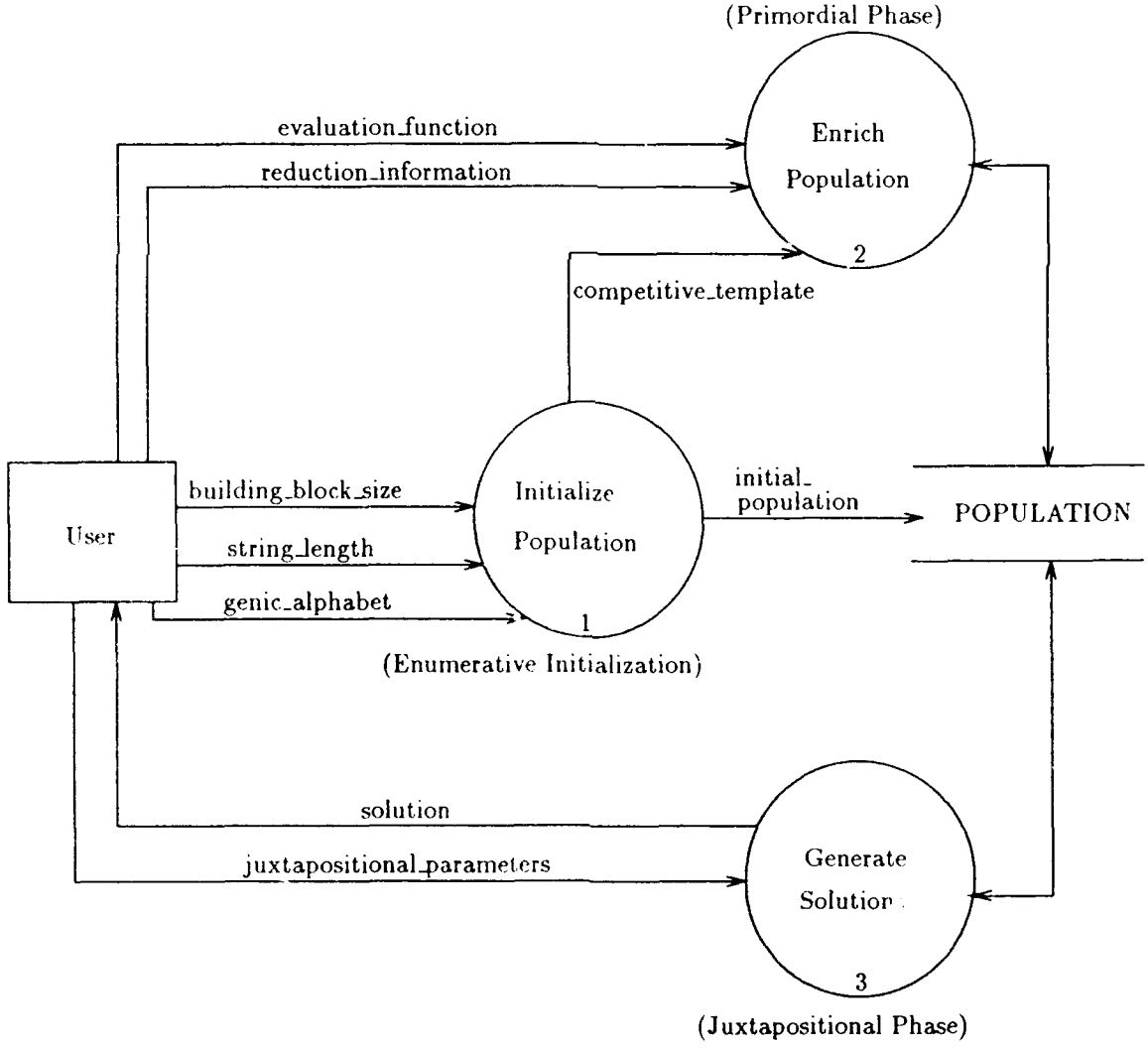


Figure 32. Level 1 Data Flow Diagram for a Messy Genetic Algorithm

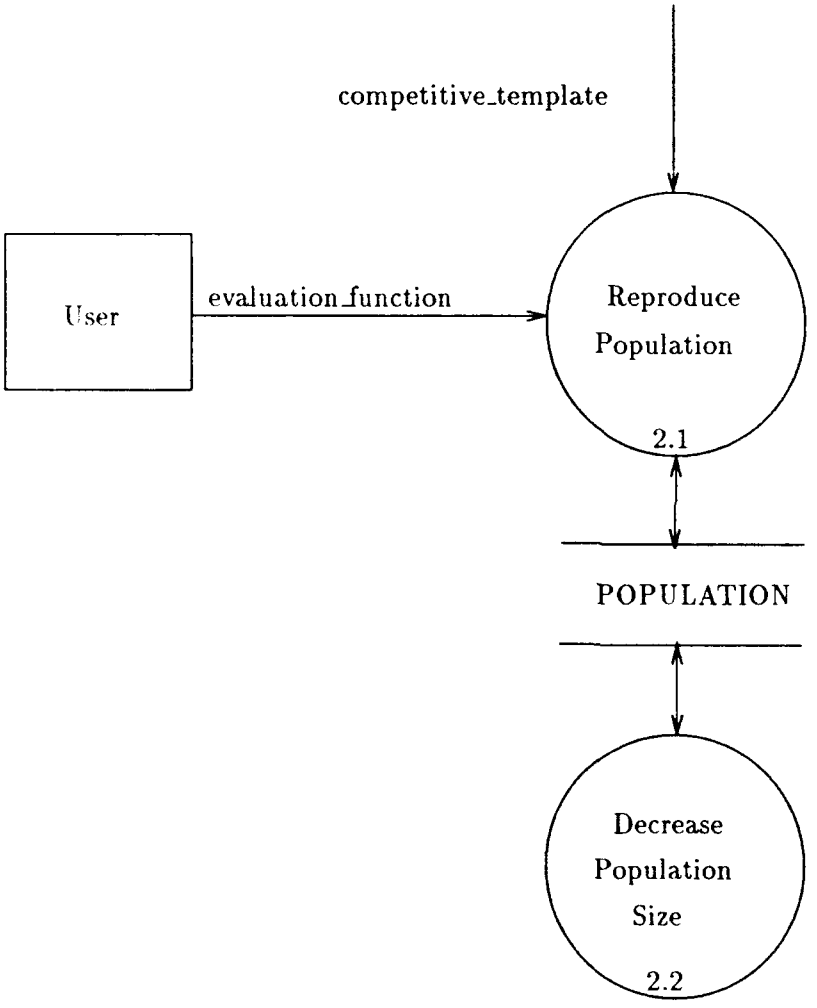


Figure 33. Data Flow Diagram for Primordial Phase

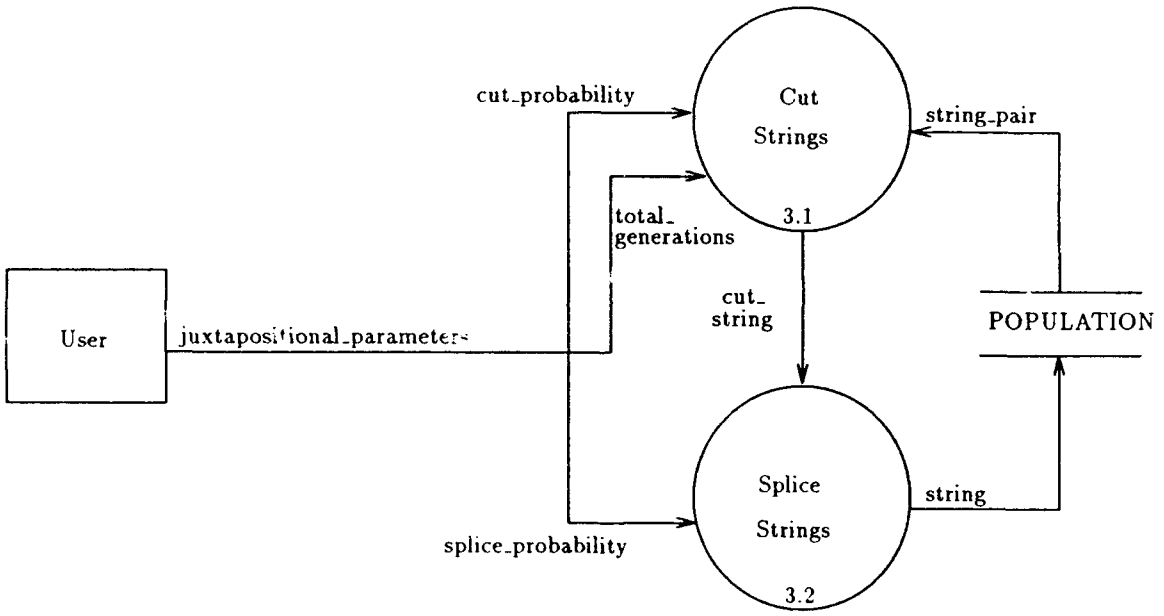


Figure 34. Data Flow Diagram for Juxtapositional Phase

Data Element Name		*Definition*
allele	=	*the value of a gene* character from the genic_alphabet
building_block_size	=	*the "highest order suspected nonlinearity suspected in the subject problem (36:505)"* *range: 1–string_length*
competitive_template	=	*a "locally optimal" string against which other strings are judged* string
cut_probability	=	*the likelihood a string will be undergo the cut operation* *range: 0.0–1.0/(*string_length*)*
cut_string	=	*a string that has just undergone the cut operation string
evaluation_function	=	*a user-supplied function that assigns a numerical value to a string based on how well the string solves the problem, i.e. f(string) → number*
gene		*atomic unit of a string* allele + locus
initial_population	=	the starting population—consists of all possible combinations of strings of length building_block_size*
juxtapositional_parameters	=	*the settings of variables for the juxtapositional phase* total_generations + cut_probability + splice_probability
genic_alphabet	=	*the characters that may be assigned to an allele – user input* **
locus	=	*the position of an gene in a string* *range: 1–string_length*
POPULATION	=	*all the strings currently existing in the genetic algorithm* 2{string}
reduction_information	=	*parameters controlling the reduction of the population* reduction_rate + reduction_interval + reduction_total
reduction_interval	=	*number of tournaments between a population reduction* integer
reduction_rate	=	*fraction by which a population is reduced in a reduction* *range: 0.0–1.0*
reduction_total	=	*total number of reduction the population undergoes* integer
solution	=	*the best feasible solution to problem found by the messy genetic algorithm* string

Figure 35. Data Dictionary for Messy Genetic Algorithm

Data Element Name		*Definition*
splice_probability	=	*the likehood two cut_strings will be concatenated* *range: 0.0~1.0*
string	=	*a candidate solution to the problem encoded using characters from the genic alphabet* 1{allele + locus} = 1{gene}
string_length	=	*the length of a string that completely and non-redundantly specifies a solution to the problem* **
string_pair	=	*two strings randomly chosen to undergo cut and splice* 2{string}2
total_generations	=	*the number of cut-splice cycles in the juxtapositional phase*

Figure 35. Data Dictionary for Messy Genetic Algorithm (cont'd)

5.4 Low-Level Design.

5.4.1 Programming Language. While the Ada programming language is available on AFIT's iPSC/2 Hypercube, the intent is to eventually port the code to different machines, such as the Intel i860 at the Vision Laboratory. Most Hypercubes do not have an Ada compiler. To have maximal portability, the C programming language, which is commonly available, is used to code the messy genetic algorithm.

5.4.2 Data Structures. Choosing data structures typically involves trade-offs between time and space. An attempt was made to choose data structures so as to use space efficiently, but also to allow the operations involving the data structures to be efficient. After evaluating the requirements of the enumerative initialization, primordial, and juxtapositional phases, no one set of data structures could be found that seemed to achieve a good balance between time and space. A closer look shows quite a dichotomy between the data structure requirements of the initialization/primordial and the juxtapositional phases (Figure 36).

Phase	Initialization/Primordial	Juxtapositional
String Length	Constant	Variable
Overspecification?	No	Yes
Fitness	Constant	Variable
Population Size	Very Large	Much Smaller
Population Changes	Decrease Only	Decrease at first, then Variable

Figure 36. Data Structure Requirements of the Different Phases of a mGA

Since the enumerative initialization and primordial phases complete before the juxtapositional phase starts, data structures for the population could be tailored to the requirements of the particular phase. As is discussed shortly, such tailoring allowed for greater time and space efficiency than if one data structure had been used. The cost was a conversion between data structures between the phases. A discussion of the considerations leading up to the selection of the data structures used in the mGA is contained in the next section.

5.4.3 Population Member. To represent a population member, some means to represent the string (gene) in terms of its loci (positions) and alleles (values) was deemed essential. Other information needed on the population member would be its fitness (for selection) and its length (determines cut probability).

5.4.3.1 Enumerative-Primordial Phase. In the enumerative phase, short, fixed-length building blocks are created. No changes are made to the length of the strings during selection in the primordial phase. The only operation that might be performed on the string is a copy to create the new population. Either a linked-list or array could have represented the string. But it was noted that in the absence of operations that would modify the string, the string would simply be a "place-holder" of information. So neither structure would offer any time efficiency over the other. The focus next turned to space efficiency, a very important consideration. Given the tremendous number of population members generated during enumerative initialization (see Section 5.4.4), a slight difference in the size of a population member would greatly influence the overall space consumed. Size efficiency definitely favored the use of an array-type structure over a linked list which would have required a pointer for each locus-allele pair.

Two array-type structures were considered for the loci. One structure would have used a binary string equal in length to a complete string. To represent building-block, the binary bits at the associated positions in the binary string would be set equal to 1, and all other bits set equal to 0. Notice such a structure would be feasible only in the initialization and primordial phases since a binary string would be unable to represent the redundant loci (overspecification) permitted permitted in the juxtapositional phase. The other structure uses an integer array equal to the building-block length, with each array element set equal to a locus position.

A choice of one of theses data structure over another was difficult because depending on the string length and building block size, either structure could be more space efficient than the other.

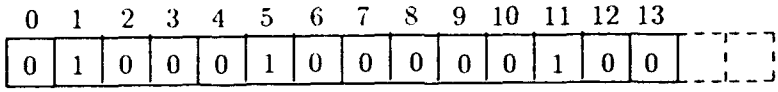
Specifically, the binary approach is more space efficient for short strings, while the integer array approach is more favorable for long strings (Figure 37).

The integer array approach was chosen because, everything else being equal, the initial population associated with a long string is larger than the initial population associated with a shorter string. This is so because for $l_1 > l_2$, $C(l_1,a) > C(l_2,a)$, where the l_i are string lengths a is the building block size. So the different in size between the two representations is multiplied more times when a long string is used than when a short string is used. As most genetic algorithms are encoded with binary strings of at least length 20, it was decided to use the integer array representation.

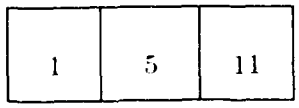
As for the alleles, the initial inclination was to use a binary string, the typical means of encoding a solution in the standard genetic algorithm. From a space efficiency standpoint, use of a binary string was very attractive. On the other hand, use of such a representation would prevent future experimentation with non-binary alphabets. Admittedly, binary alphabets are the most widely used. One notable class of exceptions are permutation problems. For example, the Traveling Salesman Problem, typically has as a genic alphabet a set of decimal numbers, each representing a city. While the decision was difficult, to allow maximal flexibility in the choice of a genic alphabet, initially a string of characters was tried. It was hoped other space saving measures would make such a representation feasible.

As for the fitness and length of the string, the decision to include or exclude them was based on their expected utility. Since evaluating the fitness of the string is a fairly expensive operation, fitness values do not change (during the primordial phase), and fitness values were needed for selection, it was decided to include a field for the fitness. Doing this prevented the continuous recalculation of a constant. As the cut operation is not performed until the juxtapositional phase, a length field would have been of no practical value, so it was excluded.

5.4.3.2 Juxtapositional Phase. Periodic reductions in population greatly reduce the size of the population by the time the juxtapositional phase is reached. For example, Goldberg's



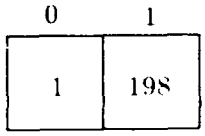
2 bytes



12 bytes

Short String
Advantage–Binary String

25 bytes



8 bytes

Long String
Advantage–Integer Array

Figure 37. Alternate Loci Data Structures

juxtapositional populations ranged from 1/16th (36:514) to 1/80000th (37:438) the size of his initial populations. Despite the significantly smaller populations, attention has to be given to memory utilization, since now the strings generally grow in length.

Even with the expected strain on memory resources, a linked-list was seriously considered for the juxtapositional phase string data structure. Initially, a linked list seemed very amenable to the splice and cut operations. Breaking links and concatenating the resulting linked lists to other link lists was thought to be a very natural model of cut and splice. Use of a linked list seemed attractive from an efficiency standpoint as well. If actual splicing and cutting are performed, the time complexity of the splice operation is O(1) with a linked list versus an $O(l_1 + l_2)$ for an array an array (l_1 and l_2 are the lengths of the two strings being spliced).

Yet further refinement of the algorithm showed that actual cutting and splicing of existing strings would not occur. To increase the genetic material needed for longer strings, cut and splice without replacement is called for. When two strings are selected for splice and cut, copies of their genetic material (alleles and loci) are made and arranged as dictated by cut and splice. When cutting and splicing are performed in this manner, the time complexity is the same whether a linked list or array is used. Since the linked list requires additional memory for pointers that an array does not, memory considerations dictated the selection of an array as the data structure for the juxtapositional string.

Once an array structure was chosen for the string, the question became how and when to allocate memory for the strings. The original inclination was to dynamically allocate memory for the strings on as "as-needed" basis. However, with the high probability for string lengthening during cut and splice, additional memory is needed very often. As dynamic memory allocation can be quite expensive in terms of time (59:372-373), a scheme which requires only an initial memory allocation was chosen.

To use this scheme, the user must specify an overflow factor. The overflow factor is multiplied by the length of a fully-specified string to give the maximum length of a string in the juxtapositional phase. At the start of the juxtapositional phase, two populations are created having arrays whose string lengths are equal to the maximum length (only a fraction of the space is used initially). One population is used as the current population whose members are subject to cut and splice. The results of cut and splice are placed in the other population, thereby creating a new population. As enough memory has been allocated to accommodate a full population of strings at their maximum lengths, no further dynamic memory allocation is needed. If a splice operation should result in a string greater than the maximum, the string is truncated.

One might question the truncating of solutions. But a string is evaluated by a left-to-right scan, with only the first instance of a gene being used. If the overflow factor is set to say, 1.6, and the problem length 30, the maximum string length is 48. It seems quite unlikely that much is being lost by limiting the string length to 48. Most, if not all genes, beyond this length would be ignored anyway. The reason for the shift in emphasis from space to time complexity is discussed more fully in Section 5.4.6.

Including a field for fitness in the string record was deemed quite important. Otherwise, 2^*P_j fitness evaluations would have to occur during selection, where P_j is the size of the population during the juxtapositional phase. Since cut and splice change the string length so often, the excessive bookkeeping needed to maintain a length field was deemed undesirable, so a length field was not included.

5.4.4 The Population Data Structure.

5.4.4.1 Dynamic vs. Static Memory Allocation. Since the string length, building-block size, and genic alphabets to be used must be decided on before running the the genetic algorithm, one strategy might be to used static data structure for the population. Constants for

these parameters could be set by the user, the program recompiled, and static data structures set up upon invocation of the program. The *run time* of a program using static memory allocation could very well be faster than a strategy which dynamically allocates memory during program execution.

There are definite drawbacks to the static allocation strategy. Since change in one of the parameters would affect the size of the population, the program would have to be edited and recompiled each time a change was made. Change is very likely, too. Different applications typically have different string encodings resulting in different string lengths. Additionally, experimentation into the affect of modifying building-block sizes and cardinality genic alphabets would be hindered by the recompilation requirement. Finally, as is discussed shortly, the initial population, while initially very large, decreases substantially throughout the primordial phase of the mGA. Since the memory apportioned to a static object remains allocated until program termination (29:35-36), memory use would be inefficient.

A dynamic memory allocation scheme alleviates such difficulties. For a given application, building-block size could be varied without recompilation. Should the encoding be changed due to a change of application or use of new genic alphabet, only the module that evaluates the string need be recompiled. Additionally, since the space assigned to the large initial population can be reclaimed when it is no longer needed, memory use is more efficient (66:181).

Since extensive experimentation, application to diverse applications, and large, memory intensive initial populations were expected, the dynamic memory allocation scheme was chosen. In C, both arrays and linked list structures can be dynamically allocated. As an abstract list is typically implemented using one of these two data structures, they were both considered as candidates.

5.4.4.2 Array vs. Linked List. Two data structures were considered for the population of solutions—an array and linked list. The requirements did not justify the use of a more

extravagant, "space-hungry" data structure, such as a tree. The following considerations drove the decision to use an array:

- No advantage to incremental population creation. With a linked list, memory could be allocated one population member at a time, while with an array, the memory for the population must be allocated all at once. If the tournament selection was *without* replacement, the incremental capability of the linked-list would be an advantage. Taking a population member from one population to another would allow memory use to remain approximately constant. But this is not the case—tournament selection occurs *with* replacement. So population members in one generation cannot be removed until the next population is completely filled. So whether a linked list or dynamic array is used, at the end of a tournament, you have two complete populations.

- No need to insert population members n any particular order. As there is no requirement to keep the list, say, ordered in ascending order of fitness, there is no benefit in using a link list over an array. Had there been such a requirement, using an array would have meant a costly shift of array elements to insert solutions in their proper order.

- After the creation of the initial population, there is no potential for a larger population. No mechanism exists for population growth during the juxtapositional phase. Instead, the population is decreased at regular intervals (Goldberg halved the population every other generation), for example (36:514). So there is no advantage to the capability of extending a linked list indefinitely (until heap space runs out). Using a linked list, memory could be recouped as the population decreases, while with an array empty population slots would be left vacant.

  The reason for doing this

- The array uses less space at the maximum population size. Enumerative initialization creates a population of $g^k * C(l, k)$, where g is the cardinality of the genic alphabet, k is the building

block size, l is the length of a fully-specified encoded function, and $C(l,k)$ is the number of combinations of l objects taken k at a time. For even relatively small values of these parameters, the initial population size will be quite large, straining the memory resources of the target computer. In fact, in a problem requiring a large initial population, Goldberg was forced to break up the population into subpopulations requiring separate primordial phases (37:438), presumably because of lack of memory resources. To avoid having to do this, the data structure for the population should be as space efficient as possible. Using a linked list, each population member would have a pointer to the next population member. An array only has a pointer to the start of the array, allowing a substantial space savings. For example, in a relatively simple problem having a string length of 30, a binary alphabet, and nonlinearity (building-block size) of 3, an initial population of 32,480 is required. On most machines, a pointer occupies about the same amount of space as an integer (50:102). Assuming 4 bytes per integer, the linked list structure would require $(32480 - 1) * 4 = 129.9$ kB more space than the array structure. So using an array would extend the size of the initial population that could be used before resorting to the undesirable subpopulation scheme.

- Use of array allows the efficient use of an indexing scheme in reproducing the population. This indexing scheme is discussed in the next section.

5.4.5 Minimizing Memory Use During Primordial Reproduction. It has already been discussed how an array uses less space at the maximum population size than a linked list. Memory utilization is still expected to be rather high. The upper bound of space utilization, which might be called the "highwater mark" would occur as the initial population is reproduced for the first time. At this point, the population has yet to be reduced in size, so the new population would contain $g^k * C(l,k)$ population members as well.

The first strategy considered for reproduction was to have two arrays of population members, one for the old generation, one for the new generation. Reproduction requires that fitness values

and strings be compared. So until reproduction is complete, it would be necessary to retain the space held by the initial population (Figure 38). So during reproduction, memory necessary for $2 * (g^k * C(l,k))$ population members would be needed. Once reproduction is complete, the space occupied by the initial population could be reclaimed since all information necessary for future reproductions is contained in the second population.

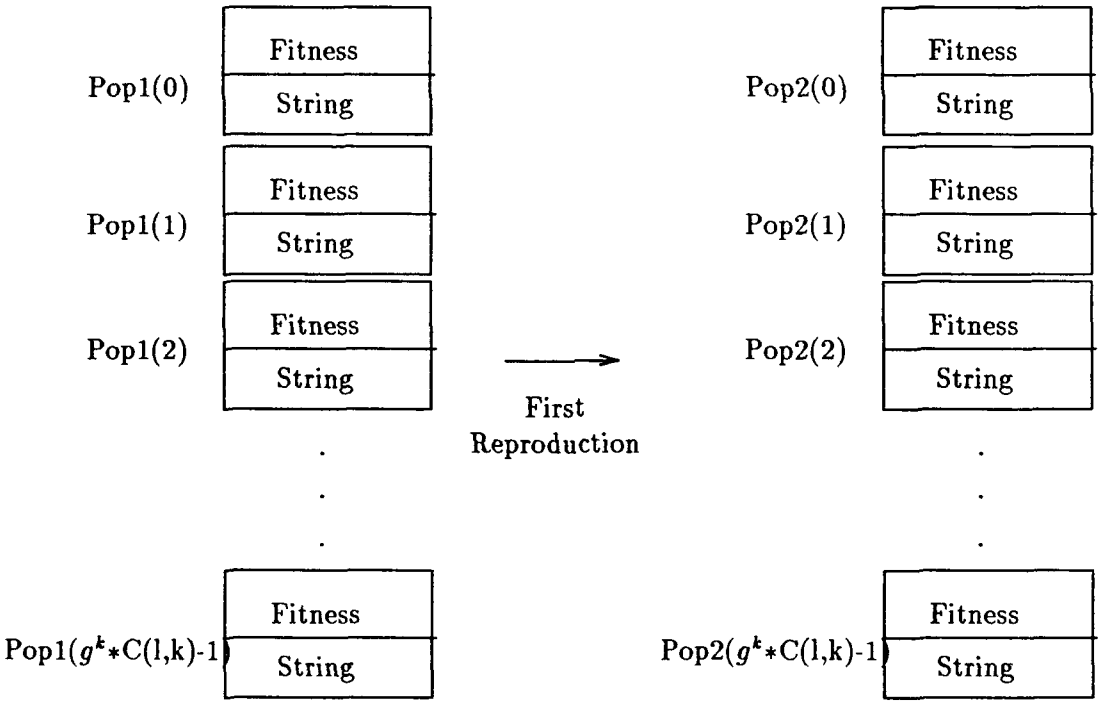


Figure 38. Population Array Reproduction Strategy – "Highwater Mark"

To help overcome the large amount of memory used by two simultaneous populations, a simple "hashing" strategy was developed. A couple of observations brought about the development of this strategy:

1. In producing a new population during the primordial phase, no changes are made to the strings.

2. Since no changes are made to the strings, their fitnesses do not change. This would not be the case if the function being optimized was probablistic, but initially these functions would not be studied.

Given these observations, it was noted that all population members in future populations could be found in the initial generation. So rather than creating a new population array, an integer array was used instead, with each integer referring to the index of a population member in the initial population.

Since the initial population is the only source of information on string fitness or structure, memory allocated to the initial population array cannot be reclaimed at the end of the primordial phase. While the initial population does consume much memory, the indexing scheme results in lower memory use at the "highwater mark." As with the first scheme, the "highwater mark" occurs in the primordial phase, before the population size has been reduced in size. In this case, though, there are three arrays at the "highwater mark"–the initial population, the first indexed population, and the second indexed population (Figure 39). The initial population is needed for string and fitness information, and the first indexed array is needed for information on the distribution. Together they provide the information necessary to produce the next population (second indexed array). Comparing Figures 38 and 39, the difference in the two schemes, then, is one population array versus two integer arrays. As each population member uses more memory than two integers, the indexed scheme is more memory efficient at the "highwater mark." In addition to saving space, the indexed scheme is likely to be more efficient in terms of time. Had the dual-population scheme been used, creating the new population would have required copying the fitness, loci, and alleles of a population member as it was inserted into the new population. With the indexed scheme, only an integer need be copied in inserting a member into the new population.

Based on the arguments given above, the indexed scheme was selected for use during the primordial phase. During the juxtapositional phase, since the cut and splice operators change the

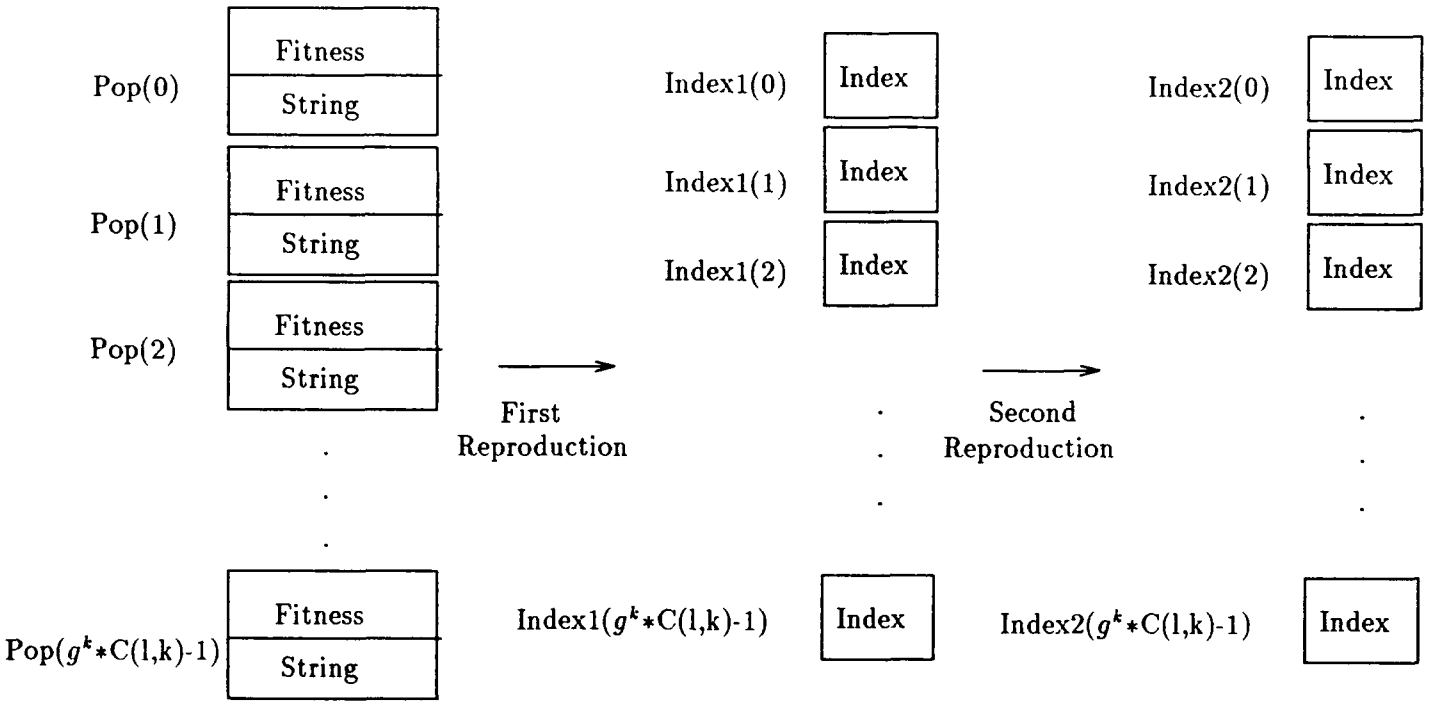


Figure 39. Indexed Reproduction Strategy – "Highwater Mark"

strings, the dual - population strategy had to be used. Initially, while strings were still short, the use of this strategy did not put a strain on the memory resources. As the strings grew (due to the cut and splice operations), checks were made to ensure the population did not exceed the heap resources. If heap space is exceeded, the population must be decomposed into subpopulations.

5.4.6 Resolving a Possible Anomaly. One may wonder why such pains (with the use of space-efficient data structures) were taken to taken to prevent memory shortages when dealing with the initialization phase, and then simply to truncate strings when they got too long in the juxtapositional phase. The reason for the care used in initializing the population is that, in the absence of heuristic knowledge that would allow the elimination of some building blocks, the developers of the messy genetic algorithm are agreed that all combinations of building blocks should be initialized (36:505). A key building block could be missed if not all building blocks are elaborated. On the other hand, it seems unnecessary to let the strings grow without limit in the juxtapositional phase when redundant genes at the ends of the strings are not considered in the evaluation.

5.4.7 Data Structure Feasibility. Preliminary tests were conducted to ensure the data structures selected would allow reasonably size problems to be initialized. The tests showed that each node of AFIT's iPSC/2 Hypercube has roughly 10 MB of heap space, which allows approximately 412,000 population to be initialized per node (3,296,000 total). Figures 40 - 42 show the memory requirements and initial population sizes (P) for three building blocks sizes and a range of string lengths (L). Notice that if a building block size of 2 is used, quite large problems can be solved using AFIT's iPSC/2 Hypercube. However its 80 Mb of heap space is quickly exceeded with building block sizes of 3 or 4. Should lack of memory resources prevent the solution of a problem on AFIT's Hypercube, either another Hypercube (or any computer) with more heap space, or a subpopulation scheme (37:438), could be used. Another alternative might be to remove the fitness field from the string data structure, and instead calculate fitness on an "as needed" basis. The additional calculations required would likely have a large impact on efficiency, but testing showed

that the elimination of the fitness field approximately doubles the size of the population that can be initialized. Notice the iPSC/2 more that adequately met the memory requirements of the classical problem ($L = 30$, block size $= 3$) used in this study.

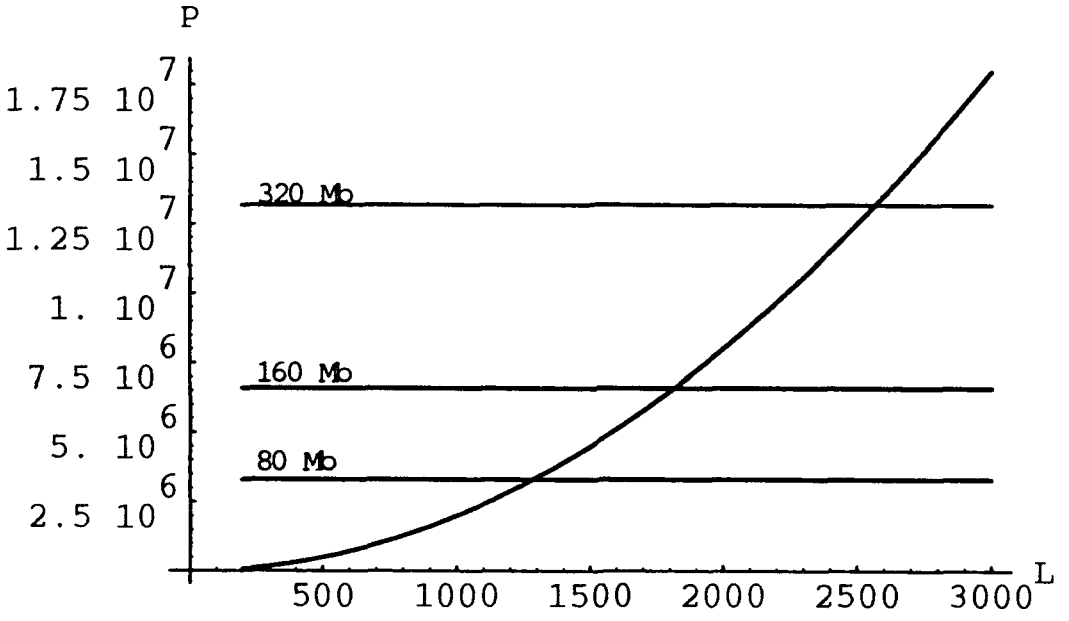


Figure 40. Memory Requirements – Building Block Size 2

5.5 Algorithm Development.

This section contains the algorithms developed to implement a messy genetic algorithm as described in the articles by Goldberg, Deb, and Korb (36) (37). Since the descriptions of many

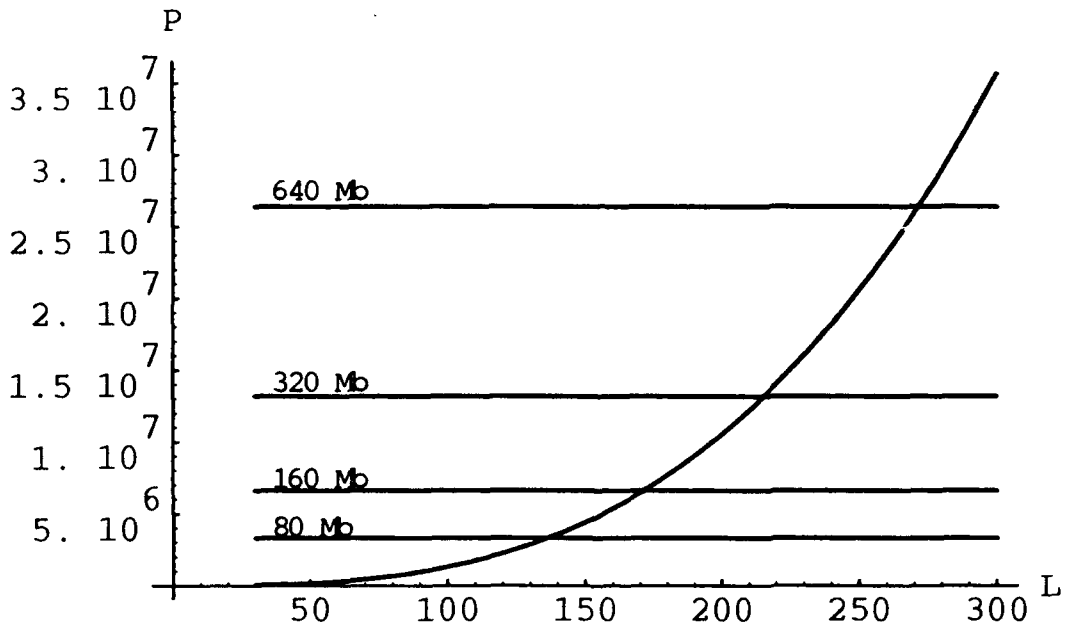


Figure 41. Memory Requirements – Building Block Size 3

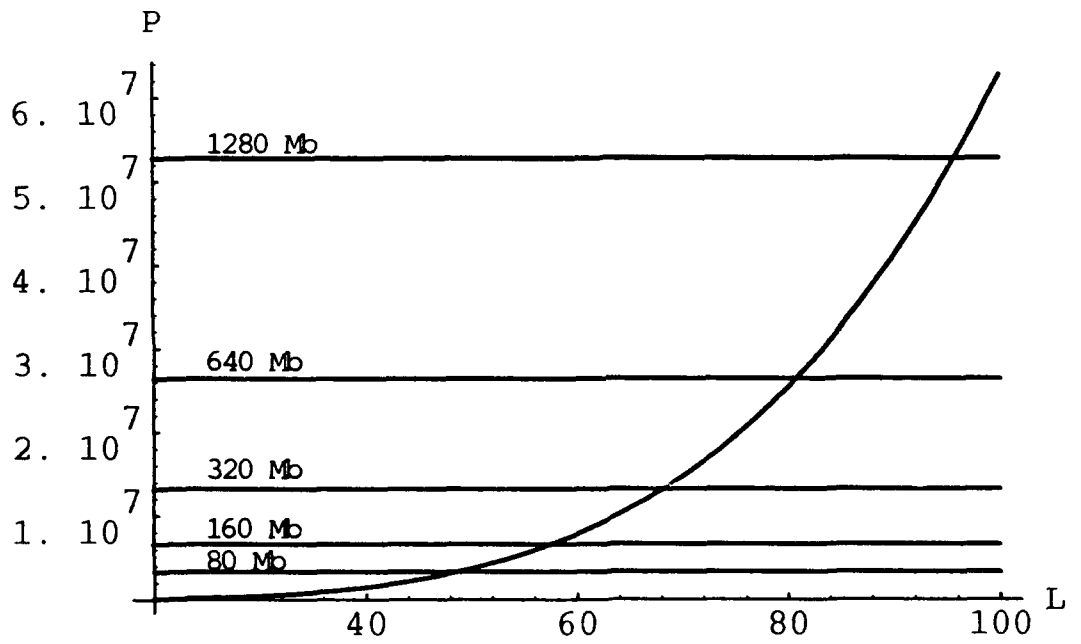


Figure 42. Memory Requirements – Building Block Size 4

of the operations are at a high level, several details had to worked out in the development of the algorithms. The validation results in the next chapter suggest the decisions made in creating the algorithms were reasonable.

5.5.1 Messy Genetic Algorithm Executive. The messy genetic algorithm is organized into three main phases—enumerative initialization, the primordial phase, and the juxtapositional phase (36:505- 506). So a functional decomposition of the messy algorithm into the three phases seemed very natural. To allow the user to change the parameters with recompiling, an input procedure was included as well. This decomposition is reflected in the executive module of the messy genetic algorithm. The algorithm is further decomposed in the sections that follow.

```
PROCEDURE Messy_Genetic Algorithm
BEGIN
   Get_Inputs(out : building_block_size, string_length,genic_alphabet);
   Initialize(in : building_block_size, string_length,genic_alphabet; out : initial_population);
   Conduct_Primordial_Phase(in : initial_population; out : population);
   Conduct_Juxtapositional_Phase(in : population; out solution);
END Messy_Genetic Algorithm;
```

5.5.2 Input Algorithm. Several several input parameters are needed to instantiate a messy genetic algorithm. Generally, only a small subset of the parameters change between runs of the messy algorithm. So it was decided to read input parameters from a file rather than prompt the user for them. The resulting algorithm is as follows:

```
PROCEDURE Get_Inputs(out : building_block_size, string_length,genic_alphabet);
BEGIN
   Open Data File;
   Read building_block_size;
   Read string_length;
   Read genic_alphabet;
   Read max_generations;
   Read juxtapositional_parameters;
   Read reduction_information
   Close Data File;
END Get_Inputs;
```

5.5.3 Initialization Algorithm. The Initialization procedure directs the construction of the competitive template and the initial population. It is decomposed into three algorithms. Gener-

ate_Competitive_Template performs the operation indicated by its name. A competitive template is needed to determine the fitness of an underspecified string. Creating the starting population involves involves constructing all possible combinations of strings equal in length to the building block size over the length of the string. Since this operation is quite algorithmically complicated, it is further decomposed as follows:

1. Create_Building_Blocks. This procedure generates all permutations equal in length to the building block size using the characters from the genic alphabet. The set of permutations is called the building_blocks.
2. Create_Initial_Population. From the set of all positions (loci) in a fully specified solution, this procedure generates all combinations of positions having *building_block_size* members. Distributing the set of building blocks over each combination results in the initial population.

The following algorithm reflects the decomposition discussed above:

```
PROCEDURE Initialize(in : building_block_size, string_length,genic_alphabet; out : population);
BEGIN
   Generate_Competitive_Template(in : string_length, sweeps, genic_alphabet);
   Create_Building_Blocks(in : building_block_size, string_length,genic_alphabet; out : building_blocks);
   Create_Initial_Population(in : string_length,building_blocks; out : initial_population);
END Initialize;
```

5.5.3.1 Generate_Competitive_Template Algorithm In the standard genetic algorithm, each string fully specifies a solution to a problem. To evaluate the fitness of a string, one need simply decode the string and examine the quality of the solution returned. Evaluating fitness in a messy genetic algorithm is not so straightforward. In the primordial phase and throughout most of the juxtapositional phase, the strings do not contain a full complement of genes, making direct evaluation impossible. Randomly generating the missing genes was found to be "too noisy to detect small signal [fitness] differences reliably" (36:518-521).

Use of a locally optimal solution, known as a competitive template, as a basis of comparison between strings, alleviates the noise problem. To evaluate a string, the genes from the string temporarily replace the corresponding genes in the competitive template. If the resulting solution has a greater fitness than the locally optimal solution, the string causing the increase in fitness

"must be a building block" (36:522). This ability to distinguish between building blocks and inferior partial solutions is especially important in the primordial phase.

No specific restrictions on the creation of a competitive template were made. Any method that can generate a locally optimal solution can be used as a source of the competitive template. Even the use of the standard genetic algorithm was suggested (36:523). For speed and simplicity, a greedy method (36:522), was chosen. In this greedy algorithm, a initial solution is randomly generated. To improve the solution, the the character (allele) values at locations along the string are varied in random permutation order. The process of generating a random permutation and alternating characters to improve the string is repeated *sweeps* times in the algorithm:

```
PROCEDURE Generate_Competitive_Template(in : string_length,sweeps, genic_alphabet);
   integer locus = 0;
   character alleles[string_length];
   integer loci[string_length];
   integer local_sweeps = 0;
BEGIN
   Allocate enough memory for competitive_template (string_length characters)
   WHILE locus < string_length
      randomly select one of the characters from the genic_alphabet
        and store in alleles[string_length];
   END WHILE;
   REPEAT
      Generate a random permutation of the positions (loci);
      FOR each of the loci in order DO
         change the value of allele[loci] to the other characters in the genic alphabet;
         keep the character that results in the highest fitness;
      END FOR;
      local_sweeps = local_sweeps + 1;
   UNTIL local_sweeps = sweeps;
END Generate_Competitive_Template;
```

Note: competitive_template is a global_variable. This is done because several other procedures need to refer competitive_template, but none of them change the value of the competitive_template.

5.5.3.2 Create_Building_Blocks Algorithm. Create_Building_Blocks generates all possible building blocks of length building_block_size. For example, with a binary alphabet and a building_block_size of 3, the output is the following array of strings:

000

001

010

011

100

101

110

111

A couple things to notice:

- The generation of building blocks involves the generation of all permutations (with repetitions) of characters in the genic alphabet. By the rule of product, if the building block length is b and the number of characters in the genetic alphabet is g, the number of building blocks is g^b (44:5). Above, b is 3 and g is 2, so the number of building blocks is $2^3 = 8$.
- The method of generation of the building blocks in the example lends itself to computer implementation. Notice how often the characters change in each position. Calling the rightmost position 0, the middle position 1, and the leftmost 2, we see position 0 varies every 1 time, position 1 varies every 2 times, and position 2 varies every 4 times. Observe $2^0 = 1$, $2^1 = 2$, and $2^2 = 4$. Generalizing, each position in the binary building block varies every 2^i building_block, for $0 \leq i \leq b$, with i being the position in the building block. If the alphabet was not binary, but an alphabet of cardinality g, each position would change to the next character in the alphabet every g^i building block.

Using these observations, an algorithm was developed for generating the building blocks. Array **Vary** has an entry for each building block position set equal to how often the position changes g^i. Array **Index** points to the current alphabetic character for that position. A building block is created by concatenating each character pointer to by **Index**. A counter keeps track of how many

building blocks have been created. The counter is checked against all entries in the **Vary** array. If a position needs to be varied, the corresponding entry in the **Index** array is set equal to the next character. The algorithm is as follows:

```
PROCEDURE Create_Building_Blocks(in : building_block_size, string_length,genic_alphabet; out : population);
   integer card;
   integer bbs = building_block_size;
   integer Vary[bbs];   / /* How often to vary character at a position */
   integer Index[bbs];    /* The current character at a position
   integer Counter = 0;
   integer i = 0;
BEGIN
   Determine number of characters in genic_alphabet and set = to card (cardinality);
   Allocate enough space for building_blocks = bbs^card
   WHILE i<bbs    /* Initialize carry array */
      Vary[i] = (card)^i ;
      Index[i] = 0;    /*Index to first character in genic alphabet*/
      i = i + 1;
   END WHILE
   i = 0;
   WHILE Counter< (card)^bbs
      WHILE i<bbs
         building_block[counter].string[i] = genic_alphabet(Index[i]);
         i = i + 1;
      END WHILE;
      i = 0;
      WHILE i<bbs
         IF (Counter mod Vary[i] == 0) THEN
            Index[i] = (Index[i]+1) mod (card - 1);
         END IF
      END WHILE;
   END WHILE;
END Create_Building_Blocks;
```

5.5.3.3 Create_Initial_Population Algorithm. This algorithm was the most complex to develop. Since it is also likely to be the most difficult to understand, the explanation of its development is quite detailed. Included are the general requirements, an alternative method, a high level description of the algorithm, an example of the action of the algorithm on a simple problem, and a lower level refinement of the algorithm.

Requirements. To complete the initialization of the population, it remains to distribute the building blocks created by Create_Building_Blocks over all possible combinations

(order is not important) of positions (loci) of the string equal in length to the building block size. Using terminology from discrete mathematics, if the string is of length n and the building_block_size is r, all combinations of the n loci taken r at a time must be generated. The cardinality of the size of combinations is $\frac{n!}{r!*(n-r)!}$. For a string of length 30 and a building block of 3, one combination of loci is 5, 8, 23. One alphabet permutation (building block) combined with one loci combination specify a single initial population member. Distributing the building blocks over all combinations gives the initial population.

Alternative. In generating the loci combinations, the initial inclination was to use recursion The reason recursion seemed a natural choice is that the task of generating combinations of a set can be decomposed into two smaller combination problems. For if example, to generate all the combinations of the characters in set {a,b,c,d,e} taken 3 at a time, one could append a to all combinations of of {b,c,d,e} taken 2 at a time, and then generate all combinations of {b,c,d,e} taken 3 at a time. The same decomposition could then be applied to the two smaller problems.

An examination of the recursive tree drove the decision not to use recursion. Such an examination can reveal whether the recursion would involve redundant work and wasted space (53:297-302). The recursive tree for the previous example is shown in Figure 43. The duplication of effort shown in this small problem would be magnified in the much larger problems expected with messy genetic algorithms.

5.5.3.4 Abstract Algorithm. To avoid the inefficiencies associated with recursion, an iterative algorithm was developed. It was known from the start that such an algorithm could be developed since every recurse algorithm has a corresponding non-recursive (iterative) version (53:432-433). If recursion involves redundant effort, often the iterative version can offer a large improvement in time and space efficiency (53:298-302).

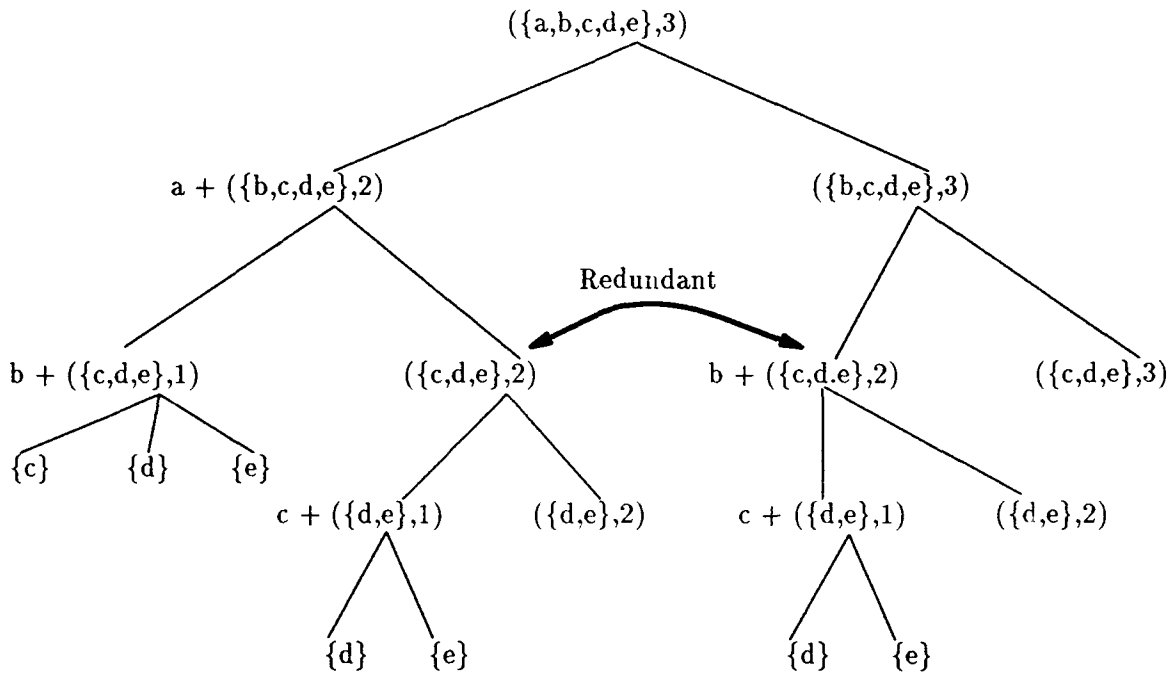


Figure 43. Recursive Tree Showing Redundant Effort

The iterative algorithm took advantage of the fact that order in a combination is not important. To make the process of generation of combination algorithmic, an order may be arbitrarily imposed. In the iterative algorithm, the positions in a combination are number from 1 to r, r being building block length, and the restriction is that each locus in a combination is less than the locus to its right. As a result of this restriction, if the building block size is r, the string length n, and the loci numbered from 1 to n, then the highest locus value at each position a in a combination is $n - r + a$. To start the process, the leftmost locus is given the lowest possible locus designation, and each succeeding locus incremented by 1. For example, in a problem of length 5, the loci might be numbered from 1 to 6. If r is 3, the initial combination would be 1-2-3 and the maximum locus values at each position are 4-5-6. The algorithm to generate the combinations may be stated as follows:

1. Record the set of numbers as a combination. If the rightmost locus is equal to its maximum value go to step 3. Otherwise, goto step 2.

2. Increment the the rightmost locus. Goto step 1.

3. If all loci are at their maximum values, terminate. Otherwise, continue with step 4.

4. Start at the leftmost locus. Check all loci to its right. If the loci are at their maximum values and the current locus is not at its maximum value, increment the current locus. Repeat for all loci except the rightmost locus.

5. Start with the locus second to the left. Reset the value of the locus to one greater the value of the locus to its left if that locus had just been changed. Repeat for all loci to the right

6. Go to step 1.

Example Problem. Consider a string with 5 allele positions and 3 building blocks. Using the above algorithm, combinations are generated as shown in Figure 44. Observe the process is similar to incrementing the low order decimal of a number then carrying to higher order decimal as overflow occurs. This iterative algorithm composes each combination immediately, rather than successively decomposing the problem as with recursion.

Algorithm Refinement. The algorithm presented in the previous section was refined to smooth the transition to the computer. For the most part, the refinement involved the addition of data structures to support the bookkeeping needed to track the locus value at each position in a combination. To reflect the realities of the programming language (C), array indexing was changed so the first element in the array is at index 0 rather than 1. Arrays **upper** and **lower** specify the highest and lowest locus, respectively, that may be assigned to a position in a combination. Each array is equal in length to the building block size. The algorithm is as follows:

```
PROCEDURE Create_Initial_Population(in : block_size, in : string_length,out : population);
    integer card;
    long popindex = 0; /* The index of the population */
    long comb(); /* comb(x,y) = # of combinations of x taken y at a time */
    integer upper[block_size];
    integer lower[block_size];
    integer loci;
```

1-2-3 (Start) Combination 1
1-2-4 (Step 2)
1-2-4 (Step 1) Combination 2
1-2-5 (Step 2)
1-2-5 (Step 1) Combination 3
1-3-5 (Step 4)
1-3-4 (Step 5)
1-3-4 (Step 1) Combination 4
1-3-5 (Step 2)
1-3-5 (Step 1) Combination 5
1-4-5 (Step 4)
1-4-5 (Step 5)
1-4-5 (Step 1) Combination 6
2-4-5 (Step 4)
2-3-5 (Step 5)
2-3-4 (Step 5)
2-3-4 (Step 1) Combination 7
2-3-5 (Step 2)
2-3-5 (Step 1) Combination 8
2-4-5 (Step 4)
2-4-5 (Step 5)
2-4-5 (Step 1) Combination 9
3-4-5 (Step 4)
3-4-5 (Step 5)
3-4-5 (Step 1) Combination 10
3-4-5 (Step 3) Terminate

Figure 44. Generation of Combinations

```
    int i, j, a;
    int total = 0;
    boolean increment[block_size];
    boolean finished = FALSE;
    boolean firstflag = TRUE;
BEGIN
popsize = cardinality^block_size*comb(string_length,block_size);
WHILE not (finished)
    IF (firstflag) THEN
        FOR (i = 0; i < block_size; i++)
            upper[i] = (string_length - (i + 1));
            lower[i] = (block_size - (i + 1));
            loci[i] = lower[i];
        END FOR;
        firstflag = FALSE;
    ELSE
        increment loci[0];
        IF (loci[0] > upper[0]) THEN
            increment[0] = TRUE;
        ELSE
            increment[0] = FALSE;
        END IF;
        FOR i = 1 to block_size - 1
    /* Increment only if adjacent position needs to be incremented */
            increment[i] = ((loci[i] = upper[i]) AND increment[i - 1]);
        END FOR;
    /*Increment the locus values as indicated by increment*/
        FOR i = block_size - 1 DOWN TO 1
            IF (increment[i - 1]) THEN
                increment loci[i];
            END IF;
        END FOR;
    /*Ensure each locus at index i is greater that the locus at i+1 */
        FOR i = block_size - 2 DOWN TO 0
            IF (loci[i] > upper[i]) THEN
                loci[i] = loci[i + 1] + 1;
            END IF;
        END FOR;
        IF (loci[block_size - 1] < upper[block_size - 1]) THEN
            finished = FALSE;
        ELSE
            finished = TRUE;
        END IF;
    END IF;
    increment total;
/*****************Distribute building blocks on the combination*********/
    FOR i = 0 to total_blocks - 1
        population[popindex].allele = building_blocks[i];
        FOR j = 0 to length of building_blocks
            population[popindex].locus[j] = loci[j];
        END FOR;
```

```
        evaluate the fitness of population[popindex].locus and
          store in population[popindex].fitness;
        IF (population[popindex].fitness > Best.fitness) THEN
            SaveBest(population[popindex]);   /*save solution*/
        END IF;
        increment popindex;
    END FOR;
/*************************End Distribution*****************************/
END WHILE;
END Create_Initial_Population;
```

5.5.4 Primordial Phase Algorithm. The goal of the primordial phase is a population enriched with high-quality substrings from which it can construct solutions in the juxtapositional phase. One way of achieving this goal is to successively construct new populations by holding "tournaments" between the strings. In a tournament, two strings are compared in terms of fitness, with the more fit string being placed in the new population. The population is also reduced at regular intervals. Since a goal of this research is to reproduce the results of Goldberg, Deb, and Korb, the same method of population enrichment is used in the messy genetic algorithm developed here. It should be pointed out, though, that there is no theoretical or practical requirement to enrich the population in this manner. Other methods could probably be developed which achieve a similar effect. However, in this initial study, the method of population discussed above was used in the primordial phase algorithm:

```
Procedure Conduct_Primordial_Phase(in : initial_population;
out : population);
    integer reductions = 0;
    integer current_interval = 0;
BEGIN
    WHILE reductions<total_reductions DO
        IF (current_interval = reduction_interval) THEN
            Reduce_Population_Size by reduction_rate;
            reductions = reductions + 1;
        END IF;
        Conduct_Tournament_Selection(in : popsize;
in out : distribution[popsize]);
        current_interval = current_interval + 1;
    END WHILE;
```

Convert population member data structure from array to linked list;
END Conduct_Primordial_Phase;

5.5.4.1 Conduct Tournament Selection Algorithm. With problems having "nonuniform scaling" (37:423-426), using tournament selection without some restriction could cause high quality building blocks of lower order subfunctions to be eliminated. Consider the function $f(x, y) = 100x + y$ encoded as a binary string of length 6 with the first three bits the x parameter and the second three bits the y parameter. Assume the competitive template is 000000. Assume the substrings 001*ddd* and *ddd*111 are selected to undergo tournament selection, where a *d* indicates missing bits. Notice that 001*ddd* is a very poor building block relative to other possible building blocks specifying x, but *ddd*111 maximizes the y component. In terms offering a prospect of eventually constructing an optimal solution, then, *ddd*111 should be perpetuated (win the tournament). However, due to the differences in scaling, when the partial solutions are overlaid on the competitive template (yielding 001000 and 000111, respectively), decoded, and evaluated, 001*ddd* wins the tournament (since f(2,0) > f(0,7)) and is placed in the new population.

To alleviate the problems associated with nonuniform scaling, a restriction that the substring must have a *threshold* number of genes in common before undergoing a tournament (37:423-426). The (arbitrary, but reasonable) choice was made that the threshold value should be equal to the upper bound of the expected number of common bits between two random substrings. For a problem of encoded length l and two strings having effective lengths (the number of nonredundant genes) of λ_1 and λ_2, the threshold is calculated from $\lceil(\lambda_1 * \lambda_2)/l\rceil$ (37:426). Having selected the first string to participate in a tournament, the question remains how many other solutions should be considered in searching for a "compatible" mate. The number of solutions considered in search of a mate is known as the shuffle number (37:424,427). Probabilistic calculations were again used to show that a shuffle number equal to the string length l offers a "reasonable probability" of finding

a compatible mate (37:426-427). Again, it should be pointed out that there is no necessity to use to use the values of the threshold and shuffle number mentioned here. These values were used in this study because they were calculated using probability theory, seemed reasonable, and resulted in optimal solutions. Additionally, a goal of this research was to first duplicate the original results before attempting anything new.

The following algorithm for tournament selection includes the threshold restriction:

```
Procedure Conduct_Tournament_Selection(in : popsize, shuffle_number;
out : distribution[popsize]);
    integer distribution[popsize];
    integer firstflag;     /*Will be a static variable in C*/
    integer permutation[popsize];
    integer shuffle_number = string_length;
    integer threshold;   /*good assumption for primordial phase is 1 */
    integer popindex = 0;   /*index of new population distribution array*/
BEGIN
    Generate a random permutation of the popsize members and store in permutation array;
    WHILE popindex < popsize DO
        Pick a population member (= cand1) at random (without replacement) from permutation array ;
        i = 0;
        candidate_found = false;
        WHILE i < shuffle_number and NOT candidate_found DO
            Pick next population member (= cand2) from permutation list;
            threshold = ceiling((effective length cand1)*(effective length cand2)/string_length);
            IF cand1 has at least threshold loci in common with cand2 THEN
                candidate_found = true;
                IF fitness of cand1 > fitness of cand2
                    Distribution[popindex] = cand1;
                ELSE IF fitness of cand1 < fitness of cand2
                    Distribution[popindex] = cand2;
                ELSE IF effective_length cand1 ≤ effective_length cand2
                    Distribution[popindex] = cand1;    /*Fitness equal, select shorter string */
                ELSE
                    Distribution[popindex] = cand2;
                END IF
            ELSE
                i = i + 1;
        END WHILE
        IF NOT candidate_found THEN    /*No second candidate found*/
            Distribution[popindex] = cand1;
        END IF
    popindex = popindex + 1;
    END WHILE
END Conduct_Tournament_Selection;
```

5.5.5 Juxtapositional Phase Algorithm. In the juxtapositional phase, the operators cut and splice are applied to the enriched population to construct solutions to the problem. Tournament selection is still used to give additional population slots to more promising solutions. The following algorithm embodies the operations involve in the juxtapositional phase:

```
Procedure Conduct_Juxtapositional_Phase(in : population; out : solution);
   integer current_generation = 0;
BEGIN
   WHILE current_generation<max_generations
      Cut_and_Paste_Strings(in out : population; out : best_in_generation);
      IF best_in_generation is better than solution THEN
         solution = best_in_generation
      END IF
      current_generation = current_generation + 1;
   END WHILE;
END Conduct_Juxtapositional_Phase;
```

5.5.5.1 Cut and Splice Algorithm The operators cut and splice are to the messy genetic algorithm what the crossover operator is to the standard genetic algorithm—they form new solutions. The goal is to create increasingly optimal solutions by successively combining the enriched building blocks from the primordial phase. Like crossover, they are binary operators, offering the prospect of creating new strings from two strings chosen as mates. Cut and splice were developed to allow strings of unequal length to be mated (36:503-504) (something that the standard crossover operator does not account for).

The first step in cut and splice involves the random selection of two mates from the population. Each string (mate) is then cut (partitioned into two substrings) with probability p_c at a random location. The cut probability is is directly proportional to the length λ of the string (36:503):

$$p_c = p_\kappa(\lambda - 1)$$

where p_κ is the constant "bitwise cut probability."

After the cut operation has completed, four possibilities exist: both strings were cut, the first string only was cut, the second string only was cut, or neither string was cut Figure 45, adapted from Goldberg's discussion of cut and slice coordination (36:503), shows a original pair of strings and the four possible sets of substrings.

The cut strings are then paired up and spliced (concatenated) with probability p_s. For the case in which both strings were cut, Goldberg specified the pairings shown below the original pair in Figure 45 (36:503). Notice Goldberg does not consider all possibilities, for instance, pairing substring 1 to 3 or reversing substrings 1 and 2. He does not justify why he only considers a subset of the splicing probabilities. One possible reason is splicing in the order shown is quite algorithmic. If the first splice does not occur, the string at the head is added to the new population, and the next possibility is considered. If the first splice occurs, the resulting string is added to the population, and the next possibility is skipped. This process continues until no more pairings exist. Notice the symmetry as well. The substring that is at the tail of one pairing is the head at the following pairing. Goldberg does not specify the pairings for the 3 cases in which both strings are not cut, but the same pattern used in the first may be applied. The resulting pairings are shown shown in Figure 45.

The following algorithm was developed to implement cut and splice in the manner described above:

```
Cut_and_Paste_Strings(in : population; out : new_population);
integer popindex = 0;
int total_strings = 0;
boolean first_cut = false;
boolean second_cut = false;
population_member cut_array[4];
integer cut_index = 0;
BEGIN
   WHILE popindex < (popsize - 1) DO     /*indexing starts at 0 */
      Randomly select two population members from the population, and
        designate them as mate1 and mate2;
      Cut mate 1 with probability p_c = p_κ(λ_1 − 1);
      IF cut occurred THEN
         first_cut = true;
         Store head of mate1 in cut_array[0];
```

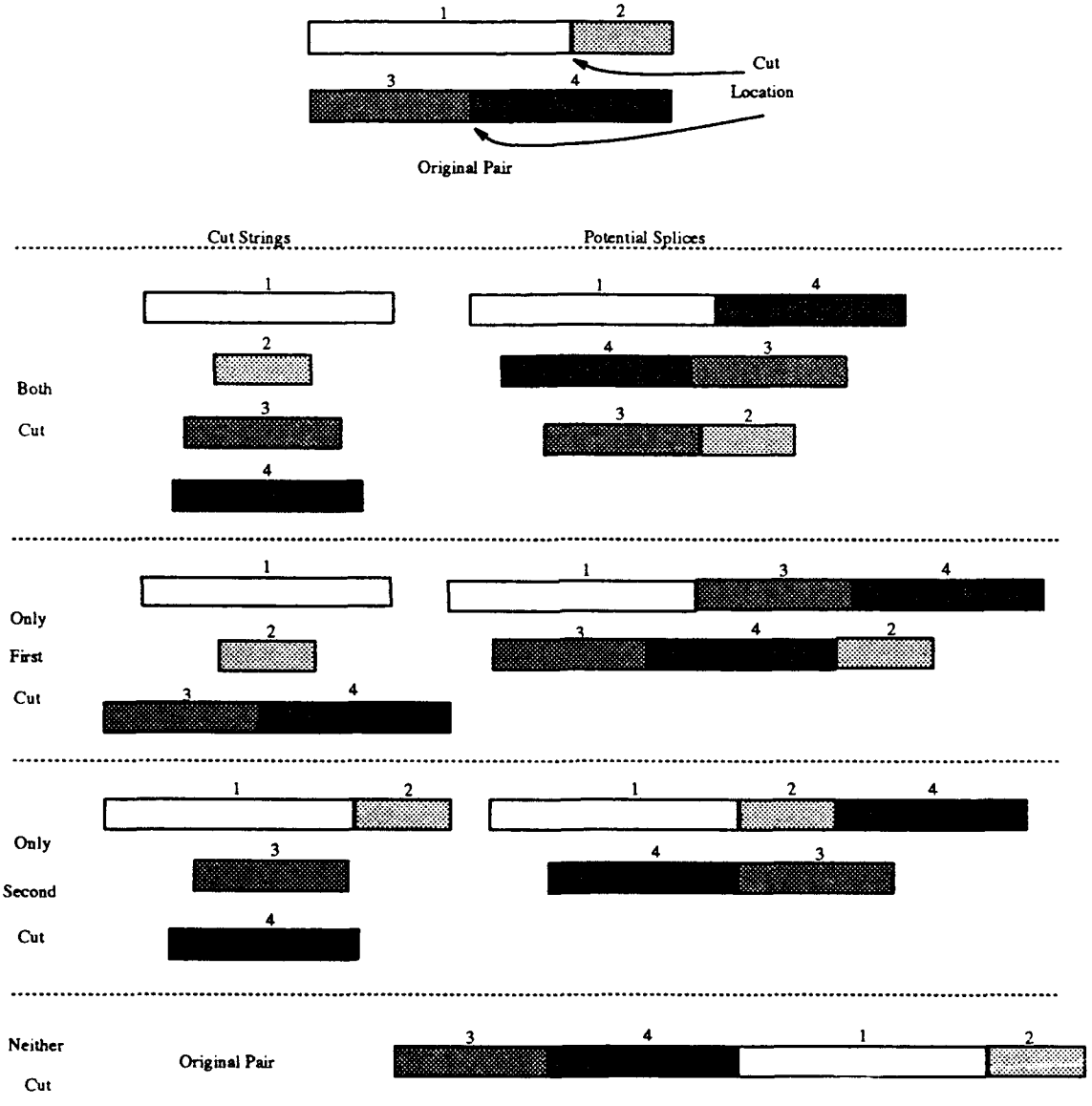


Figure 45. Cut and Splice Operations

```
            Store tail of mate1 in cut_array[3];
        ELSE
            Store mate1 in cut_array[0];
        Cut mate 2 with probability p_c = p_κ(λ_2 − 1);
        IF cut occurred THEN
            second_cut = true;
            Store tail of mate2 in cut_array[1];
            Store head of mate2 in cut_array[2];
        ELSE        /*second string not cut */
            Store mate2 in cut_array[1];
            IF first_cut THEN
                MOVE tail of mate1 from cut_array[3] to cut_array[2];
            END IF;
        END IF;
        IF first_cut and second_cut THEN
            total_strings = 4;
        ELSE IF NOT (first_cut and second_cut) THEN
            total_strings = 2;
        ELSE
            total_strings = 3;
        END IF;
        first_cut = false;    /* Reset for next time */
        second_cut = false;
        WHILE cut_index < total_strings DO
            IF cut_index = total_string - 1 THEN    /*Only 1 string left */
                Put cut_array[cut_index] in new_population[popindex];
                popindex = popindex + 1;
                cut_index = cut_index + 1;
            ELSE
                Concatenate (splice) cut_array[cut_index] and cut_array[cut_index + 1] with probability p_s;
                    IF splice occurred THEN
                        Put concatenated string in new_population[popindex];
                        popindex = popindex + 1;
                        cutindex = cutindex + 2;    /*Skip to next pair*/
                    ELSE    /* Put head in new pop. and try to pair tail */
                        Put cut_array[cut_index] in new_population[popindex];
                        popindex = popindex + 1;
                        cut_index = cut_index + 1;
                    END IF;
            END IF;
        END WHILE;
    END WHILE;
END Cut_and_Paste_Strings;
```

5.5.5.2 Conduct Tournament Selection. At an abstract level, the tournament selection algorithm for the primordial phase is identical to the selection algorithm presented in Section 5.5.4.1.

Due to the conversion of the string data structure, the algorithm had to be slightly modified (see next section).

5.6 Coding the Messy Genetic Algorithm.

The data structures and algorithms were designed in sufficient detail to allow the coding phase to be largely a syntactic map from the design to C code. There were exceptions, though. The abstract list structures used in the algorithms had to be replaced with the data structures specified in Section 5.4—arrays. Fortunately, an array notation was used to represent the abstract list in the algorithms, so mapping the list to a C array was quite natural. A few details other were deferred to the coding phase as well. The user interface is more elaborate than the algorithm indicates. To make the program more user friendly, an input form was developed for reading the genetic algorithm parameters from a file. Additionally, converting the strings from the primordial data structures to the juxtapositional data structure was not addressed until the coding phase since this operation is quite language-specific. A separate procedure called at the start of the juxtapositional phase was developed to handle the data structure conversion. Finally, a few low level procedures not documented in Section 5.5 were developed to support the higher level procedures. The structure chart in Figure 46 shows all the procedures in the messy genetic algorithm and their calling dependencies. An unfortunate ramification of converting the string data structure between the primordial and juxtapositional phases is that different versions of several procedures had to be created. The only difference between a procedures "Name" and "NameJ" in the structure chart is the string data structure on which they operate.

5.7 Test Strategy.

A program is often specified in terms of the allowable starting conditions (precondition) and the desired final conditions (postconditions) (22:8). A proof of program correctness (verification) would involve demonstrating that given the precondition the program guarantees the postcondition

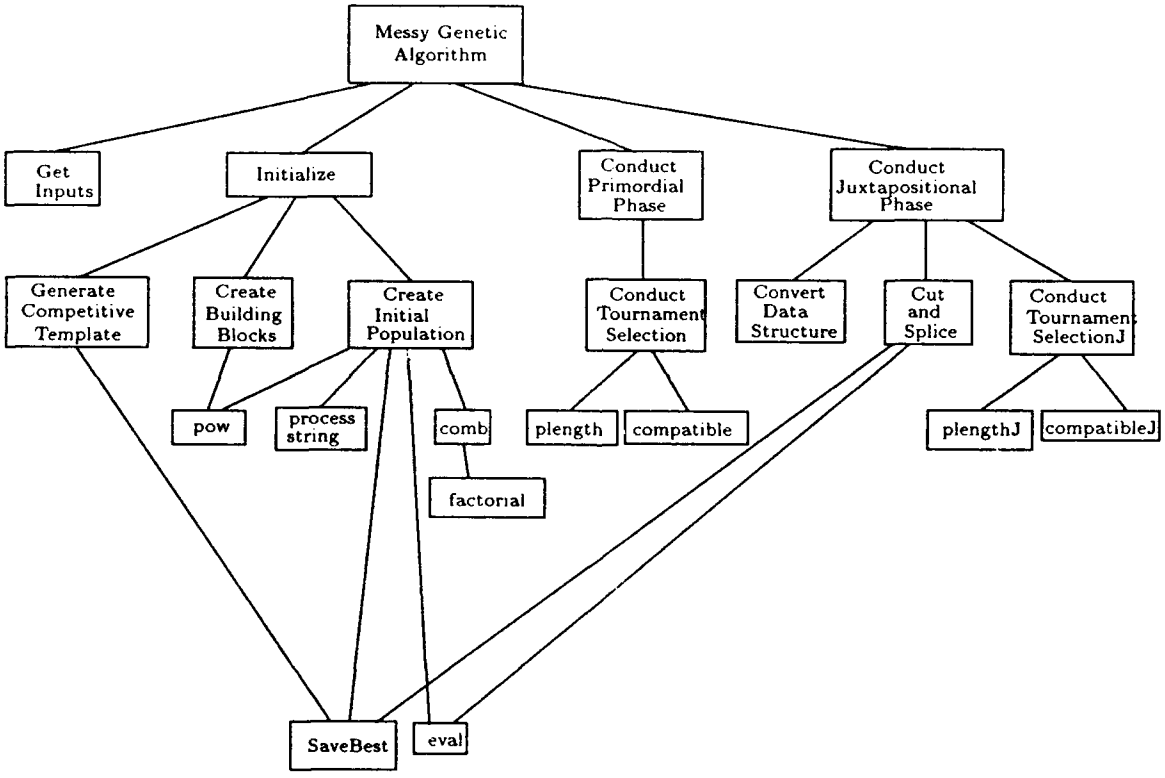


Figure 46. Structure Chart for Messy Genetic Algorithm

results. A validation of the program would involve various tests or experiments using (usually) a subset of all possible preconditions and confirming the program returns a solution meeting the postcondition.

The nature of a messy genetic algorithm makes either a verification or validation in the sense described above almost useless. Like a standard genetic algorithm, a messy genetic algorithm lacks a convergence theorem. So given a set of input parameters (the precondition), nothing specific can be said as to the output (postcondition) expected from a correct messy genetic algorithm. The postcondition might specify the solution is best solution uncovered in the source of the search, but any number of search algorithms meet this criteria. Such a vague (weak) postcondition leads to a vague sense of correctness.

As a result, the best strategy in showing a correct implementation seems to be to compare the results of the messy genetic algorithm with the published results of Goldberg, Korb, and Deb. A similarity in solutions (postconditions) at least can give some confidence that the genetic algorithm is implemented correctly. However, it should be recognized that difficult algorithms may return identical solutions for some inputs, but differing solutions on another set of inputs. So such a relative validation should be "taken with a grain of salt."

The situation is not as satisfactory with several of the operations which comprise the messy genetic algorithm. For instance, creating the initial population is a very well defined operation. The inputs are the genic alphabet, the building block length , and the string length (preconditions), and the output must be all possible combinations of the building blocks. In this case a validation could involve comparing the combinations generated by the program with a set of combinations generated manually. All components are quite amenable to a similar validation strategy.

5.8 Choice of Problem.

The first application messy genetic algorithms was against a deceptive problem developed from earlier work on GA-hard functions (32) (33). To validate the messy genetic algorithm developed in this research it was decided to to apply it to the same deceptive problem, and compare results. A description of the problem follows.

The problem is composed of ten 3-bit subfunctions, coded as a alleles on string of length 30 (36:511). Each 3-bit function is evaluated according to Figure 47, then added to obtain the overall fitness.

Alleles	Fitness	Alleles	Fitness
000	28	100	14
001	26	101	0
010	22	110	0
011	0	111	30

Figure 47. Fitness Value for the 3-bit Subfunctions

Graphs of the subfunction in Hamming space (Figure 48 (36:510)) and as a function of the bit pattern (Figure 49 (36:510)) reveal its deceptive nature. The bit pattern 111 has the highest fitness, but all the other bit patterns are coded so as to penalize a subfunction having 1's in its bit pattern. The effect of this is the isolation of the bit pattern 111, and a fitness gradient towards the second best solution 000. Combining ten subfunctions into one problem makes for a difficult functional optimization inappropriate for a gradient search technique (36:511). The optimal result for such a problem is a string of 30 1's.

Experiments with a simple genetic algorithm showed that the effectiveness of the search dependent on the location of the bits of the subfunction on a string. The simple genetic algorithm was able to find the optimal result when the bits of the subfunction were adjacent to one another (a tight ordering), but failed to find the optimal for a random ordering and a loose ordering. The messy genetic algorithm found the optimal for all three orderings (36:511-511).

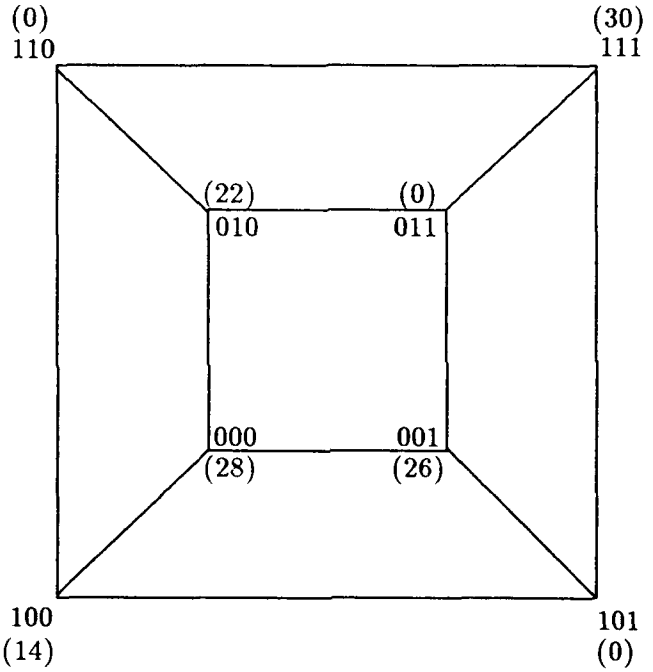


Figure 48. Hamming Graph of Deceptive Subfunction

(36:510)

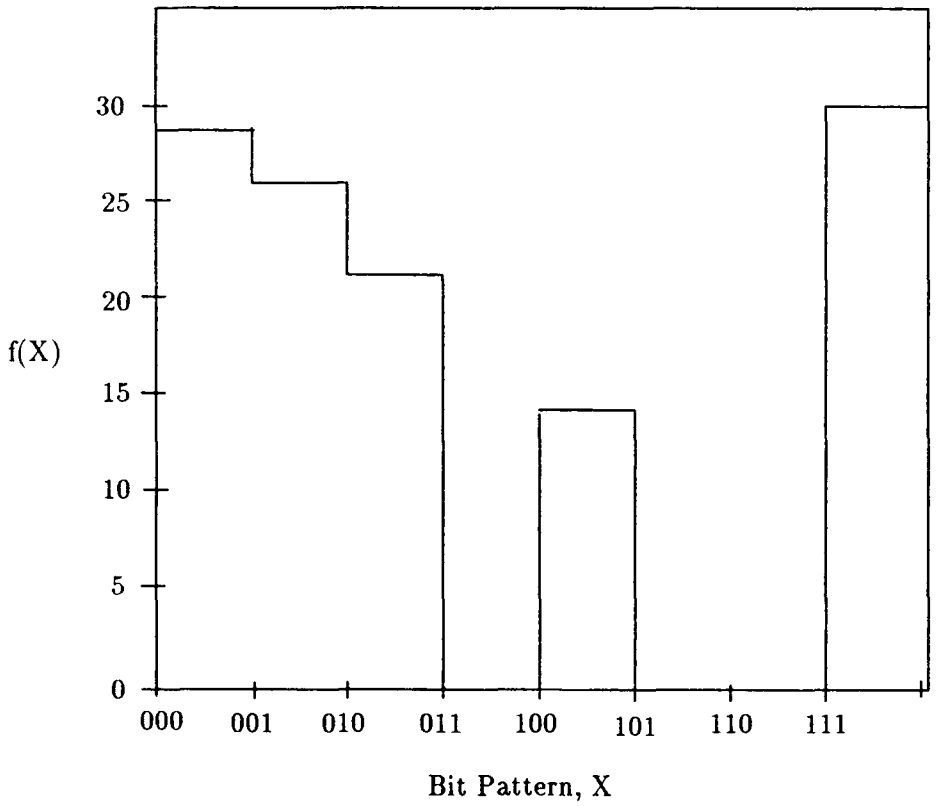


Figure 49. Fitness Gradient Leads Away from Optimal

(36:510)

Since a loose ordering provides the toughest test (36:513), a loose ordering is used to validate the messy genetic algorithm developed in this research. An examination of the loose ordering used by Goldberg, Deb, and Korb (36:511) showed one of the subfunctions is not as loose as possible. The ordering they show is 1 4 7 ... 2 5 8 ... 3 6 9 ... 24 27 30. Elaborating the details implied by the ellipses gives the following complete set of orderings:

1 4 7	11 14 17	23 26 29
2 5 8	12 15 18	24 27 30
3 6 9	19 20 21	
10 13 16	22 25 28	

One of the orderings, 19 20 21, is tightly ordered. To maximize the difficulty of the problem, it was decided to use the following set of orderings, all of which are loose:

1 6 11	5 10 15	19 24 29
2 7 12	16 21 26	20 25 30
3 8 13	17 22 27	
4 9 14	18 23 28	

5.9 Summary.

This Chapter documented the design process of a sequential messy genetic algorithm. The design documentation began with a problem description in Section 5.1, and continued in with increasing level of details through the other design steps. Section 5.2 specified the operations and high-level data structures needed to meet the requirements of a messy genetic algorithm. In Section 5.3 documented the low level design, in which specific data structures were chosen after comparing their efficiency and effectiveness with other data structures. The algorithms and their development were discussed in Section 5.4. A GA-hard problem that had been previously solved by a messy genetic algorithm was used to validate the messy genetic code resulting from the design

presented in this chapter. This problem and the messy genetic algorithm's performance against it were discussed in Section 5.5.

Despite the fact that the Genetic Algorithm Toolkit is being developed for an iPSC/2 Hypercube, the development of the sequential code was not a wasted effort. First, when the sequential messy genetic algorithm performed similarly to the messy genetic algorithm reported in the literature, confidence was gained that it salient features had been understood and captured in the design. Additionally, most of the data structures did not change or changed very little in implementing the messy genetic algorithm on a coarse-grained parallel computer.

VI. Parallelization of the Messy Genetic Algorithm.

6.1 Introduction.

This chapter documents the steps taken in parallelizing the messy genetic algorithm. Section 6.2 examines the sequential bottleneck in the messy genetic algorithm. Ideally, the bottleneck would be one ofthe first areas addressed by the parallel design. Since an operation may be inherently parallel and not a good application for a parallel computer, Section 6.3 addresses the feasibility of parallelizing the various parts of the messy genetic algorithm. The anaiysis yields a parallel decomposition with the potential for a near linear speedup of the sequential bottleneck. Encouraged by the prospect of a substantial decrease in run time, a parallel messy genetic algorithm was implemented. The mapping to the nodes of a Hypercube is discussed in Section 6.4. The implementation results are presented in the next chapter.

6.2 Sequential Bottleneck.

Figure 50 shows the run times for various phases of the messy genetic algorithm on 1 node of the iPSC/2 Hypercube computer. The times are averaged over three separate runs of the messy genetic algorithm, each using a different random seed. The optimal solution was found in each of the runs. For further details on the experiment, consult Chapter VII.

	Time (sec)	Percentage
Generate Competitive Template	0.014	0.0049
Create Building Blocks	0.001	0.0003
Initialize Population	16.708	5.840
Primordial Phase	250.699	87.634
Convert Data Structure	0.635	0.222
Juxtapositional Phase	17.903	6.258
Other	0.116	0.0405
Total	286.076	1.000

Figure 50. Time Data for Various Parts of the mGA - 1 Node

With the phases decomposed in this manner, the primordial phase definitely seems to be the bottleneck. Attacking bottlenecks is a key way of reducing the run time of a program. Since the primordial phase so dominates the messy genetic algorithm, a significant reduction in the run time of the primordial phase can result in a significant overall reduction in the run time. Use of the Hypercube may not provide the answer. Not all time-intensive applications are good choices for parallel computers. If it turns out the primordial phase is inherently sequential, parallelization of the messy genetic algorithm would offer little chance of improvement. Fortunately, the analysis of the primordial phase in the next section uncovers a high degree of parallelizability, so parallelizing the entire algorithm is worthwhile. While the other phase consumed much less of the overall run time, the feasibility of parallelizing these phases are examined as well. The results are presented in the next section.

6.3 Determining Effective Parallel Decompositions.

This section examines the feasibility of parallelizing the phases of a messy genetic algorithm. Since a Hypercube is the target machine, a coarse-grain perspective is used. Not only is likely speedup discussed, but also whether the sequential and parallel versions are functionally equivalent or not.

6.3.1 Generation of the Competitive Template. In the sequential version, a random solution is generated and a greedy algorithm traverses the string n_{sweeps} times, trying different allele values at each locus position. A data decomposition, with each node computing a portion of the template does not seem to be feasible. Since the problems applied to a messy genetic algorithm are non-linear, the portions could not be computed independently. That is, optimizing portions of a non-linear problem does not guarantee the solution as a whole is good. Additionally, little time savings could be expected in parallelizing the generation of the solution. In the sequential runs, template generation is certainly not the bottleneck. Another approach is to instead try to achieve better

quality solutions instead of lower run times. To achieve this, one could place an identical greedy algorithm on each node of the Hypercube and traverse the solution n_{sweeps}/p, p being the number of nodes on the Hypercube. The nodal results could then be compared (on an executive level or using gdhigh) to determine the best template. Determining the best solution would require a relatively costly communication step resulting in a longer run time than the sequential version. As the sequential version already was returning fairly high quality solutions, it was decided to forgo parallelizing the generation of the competitive template.

6.3.2 Enumerative Initialization and Evaluation. Parallelization of this phase of the genetic algorithm certainly should be considered desirable. Creation of the initial population involves the generation of all combinations of loci taken r at a time, r being the building block length (suspected nonlinearity) of the problem. If there are n loci in a fully specified problem, there are $\frac{n!}{r!*(n-r)!}$ combinations, typically quite a large number. Additionally, each combination requires an evaluation, a fairly expensive operation.

6.3.2.1 Rejected Parallel Decomposition. Notice the example problem can be decomposed into 3 smaller problems:

- * Put 1 at the head of all combinations of 2,3,4,5 taken 2 at a time.
- ** Put 2 at the head of all combinations of 3,4,5 taken 2 at a time.
- *** Put 3 at the head of all combinations of 4,5 taken 2 at a time.

Using the iterative algorithm above, the three sub-problems are independent of each other, they could be solved in parallel. For example, one of the subproblems could be assign to nodes 0, 1, and 2.

In Section 5.5, it is shown that the generation of combinations of building blocks could be achieved with a series of smaller combinations. This is exactly what is happening with this

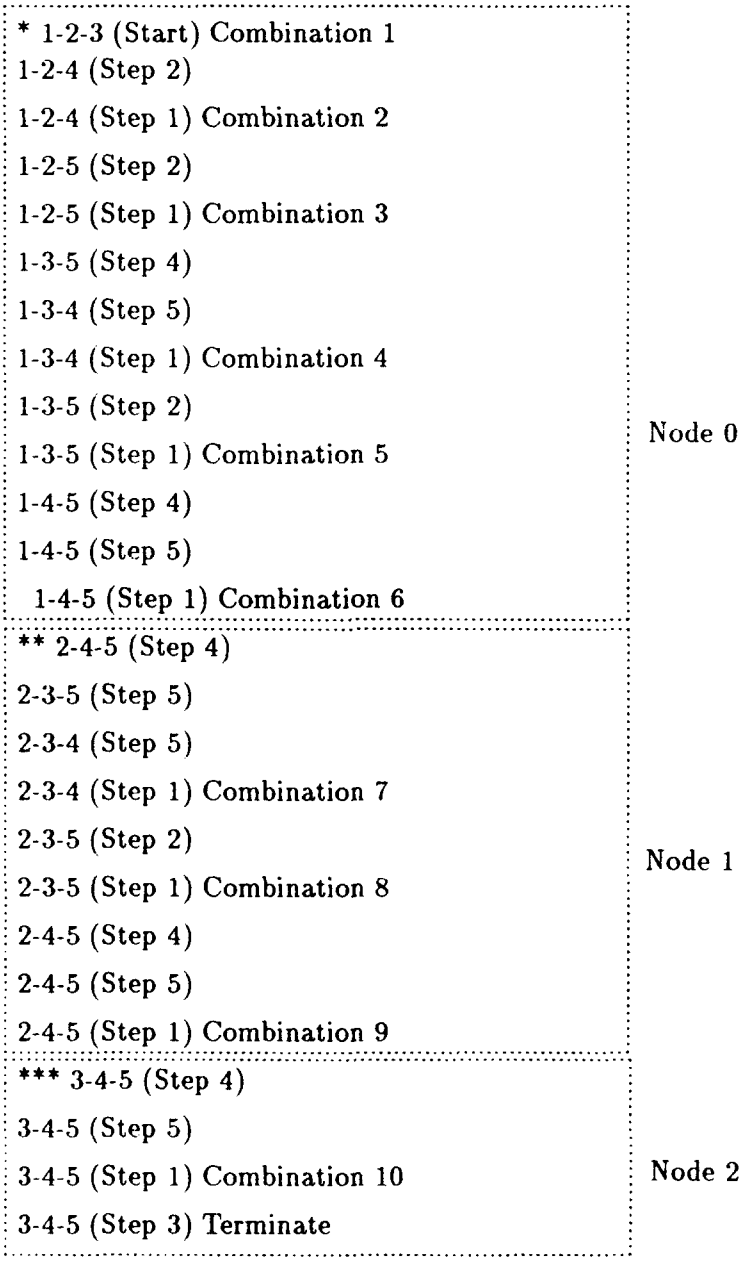


Figure 51. A Considered Parallel Decomposition

decomposition—each node is assigned a subproblem. An immediate concern with this decomposition is that of load-balance. This concern arises by noting that for a fixed r,

$$\binom{n}{r} > \binom{n-1}{r}$$

Figure 51 shows quite a large load imbalance for the small sample problem. A balancing scheme was developed that offered the potential for fairly good load balance for longer strings. Figure 6.3.2.1 show an example of this scheme for a problem of length 30 and a building block size of 3. On the first pass, the sub-problems are distributed to the nodes in ascending order of magnitude. Thereafter, the reverse is done. That is, to offset the initial load imbalance, the remaining sub-problems are distributed to nodes in descending order of magnitude. The example shows a 6.4 per cent difference (imbalance) between the node with the most combinations and the node with the node with the least combinations. While this is a fairly good load balance, this scheme was rejected when another scheme offering a near-perfect load balance was discovered. This scheme is presented next.

6.3.2.2 Load-Balanced Parallel Decomposition. Consider a list of combinations generated by a sequential implementation of the combination algorithm. The equivalent list can be generated with a near perfect load balance as follows. To begin, each node is given a starting offset into the list of combinations corresponding to the node number plus 1. For example, node 0 is assigned the first combination, node 1 the second combination, and so forth. Thereafter, each node generates the combination that is a multiple of the total number of nodes plus the starting offset. On an eight node Hypercube, then, node 0 generates the 1st, 9th, 17th, ..., node 2 generates the 2nd, 10th, 18th, ..., and so forth. The stars in Figure 52 represent the node on which the combination is generated for the sample problem. Using this decomposition, the nodes differ by at most one combination, a near-perfect load balance.

Node 0 $\binom{29}{2} + \binom{14}{2} + \binom{6}{2} = 519$

Node 1 $\binom{28}{2} + \binom{15}{2} + \binom{7}{2} = 504$

Node 2 $\binom{27}{2} + \binom{16}{2} + \binom{8}{2} = 499$

Node 3 $\binom{26}{2} + \binom{17}{2} + \binom{9}{2} = 497$

Node 4 $\binom{25}{2} + \binom{18}{2} + \binom{10}{2} + \binom{2}{2} = 499$

Node 5 $\binom{24}{2} + \binom{19}{2} + \binom{11}{2} + \binom{3}{2} = 495$

Node 6 $\binom{23}{2} + \binom{20}{2} + \binom{12}{2} + \binom{4}{2} = 515$

Node 7 $\binom{22}{2} + \binom{21}{2} + \binom{13}{2} + \binom{5}{2} = 529$

1-2-3 (Start) Combination 1 *
1-2-4 (Step 2)
1-2-4 (Step 1) Combination 2 **
1-2-5 (Step 2)
1-2-5 (Step 1) Combination 3 ***
1-3-5 (Step 4)
1-3-4 (Step 5)
1-3-4 (Step 1) Combination 4 *
1-3-5 (Step 2)
1-3-5 (Step 1) Combination 5 **
1-4-5 (Step 4)
1-4-5 (Step 5)
1-4-5 (Step 1) Combination 6 ***
2-4-5 (Step 4)
2-3-5 (Step 5)
2-3-4 (Step 5)
2-3-4 (Step 1) Combination 7 *
2-3-5 (Step 2)
2-3-5 (Step 1) Combination 8 **
2-4-5 (Step 4)
2-4-5 (Step 5)
2-4-5 (Step 1) Combination 9 ***
3-4-5 (Step 4)
3-4-5 (Step 5)
3-4-5 (Step 1) Combination 10 *
3-4-5 (Step 3) Terminate

Figure 52. Load-Balanced Parallel Decomposition

6.3.2.3 Benefits of the Parallel Decomposition. In addition to a speedup, a parallel decomposition allows the initialization of much larger problems than possible with the sequential version run on 1 node. The initial population consumes a large amount of memory. In a experiment to measure heap space, each node was found to have about 10 Mb in heap space, allowing the initialization of approximately 410,000 members of a population. Distributing the initialization among the 8 nodes of the iPSC/2 Hypercube allows the initialization of 3.2 million member population. A final positive comment is the parallel version of enumerative initialization is functionally equivalent to the sequential version.

6.3.3 Primordial Phase. It seems virtually impossible to parallelize the primordial phase and maintain function equivalence in the primordial phase. In the sequential version, a tournament could be held between any two members of the population. With a distributed population, to allow such global tournaments would require numerous communications of fitness information. Such overhead would likely offset any time gains.

To allow this phase to operate in parallel, a decomposition akin to the one used by Pettey (63) for simple genetic algorithms is used. Tournaments are held on a strictly local basis for a number of generations specified by the user. Doing so results in a better than linear speedup in the primordial phase (see next Chapter), yet the possibility exists for locally optimal building blocks. This possibility is dealt with at the start of the juxtapositional phase.

6.3.4 Juxtapositional Phase. In order to construct good solutions to the problem, the best building blocks are needed. Leaving the nodal populations separate in the juxtapositional phase generally left each node lacking one or more of the best building blocks. Poorer solutions than those returned by the sequential algorithm resulted.

To overcome this problem, something quite "drastic" is done. At the start of the juxtapositional phase, the local populations are combined on each of the nodes. To further enrich the popu-

lation and reduce the number locally optimal, but globally inferior, building blocks, the combined population can undergo additional tournaments prior to the conduct of the normal juxtapositional phase[1]. It is important to note that combining the population like this does not result in identical and redundant searches. While the populations on each node taken as a set are identical, the order of the population members on each node is different. So the selection of mates for cut and splice (and hence the resulting strings) differ.

In reality, this scheme is not as "drastic" as it might seem. The resulting population size is the same as a sequential messy genetic algorithm has at the start of the juxtapositional phase. So there is virtually no difference in run time in the juxtapositional phase. True, when using a parallel computer, one normally hopes for a speedup. But there does not seem to be an alternative, in general. Sharing of the best building blocks could not be used, because of differences in subfunction scaling. Once the nodal populations reach a certain value, the building blocks could be combined on a single node for further tournaments and reductions.

One possibility not explored is assigning each node different cut and splice probabilities, allowing a broader search than a sequential mGA using one set of cut and splice probabilities. While not leading to a speedup of the juxtapositional phase, such a scheme would guarantee solutions at least as good (and quite likely better) than a single processor mGA.

6.4 Mapping to the Hypercube.

A mapping specifies the way the data and algorithms derived from the decomposition are distributed among the processors of the parallel computer (67:23,26-29). The goals of a mapping are a good load balance and scalability (67:26-27). The mapping discussed below seems to guarantee a good load balance, but due to an expected dependence of genetic algorithm performance on the

[1] Additional tournaments were needed to solve the target problem to optimality

number of nodes used, it is impossible to make assertions as to scalability. The reason for this is discussed shortly.

All parallel decompositions discussed in the last section are data decompositions. Each node on the iPSC/2 can be assigned the same set of algorithms. The data, in this case the strings, is evenly divided among the nodes of the Hypercube. Assuming each node is assigned the same cut and splice probability, run times can be expected to be virtually equal. Even if the probabilities differ, the resulting differences in run time are very likely to be small relative to the overall run time. (Cut and splice probabilities only play are part in the cut and splice operation; all other operations are independent of these probabilities). As a result, an excellent load balance can be expected.

It might seem that such a mapping is scalable. In a sense, this is true. Up until the number of processors exceeds the number of strings, the same mapping may be used. So there is a certain independence from the dimensionality of the Hypercube. However, recall that tournament selection is conducted on a local basis. As the number of nodes increases, tournament selection becomes increasing localized. Since in a sequential messy genetic algorithm, tournament selection is on a global basis, as the cube size increases, the parallel implementation diverges more and more from its sequential counterpart. The difference in tournament selection can result in different solutions. So the mapping cannot be said to be truly scalable. Testing may give some indication as to the effect of increased localization.

6.5 Summary.

This chapter began with an identification of the sequential bottlenecks of the messy genetic algorithm (Section 6.2). Section 6.3 showed that the parallelization of the messy genetic algorithm is feasible by developing a decomposition that has the potential for a substantial speedup. Section 6.4

discussed the mapping of the decomposed problem to the nodes of the coarse-grained computer. The next chapter discusses the results of the parallel implementation.

VII. Messy Genetic Algorithm - Implementation Results.

7.1 Introduction.

This chapter presents the implementation results of the sequential (Section 7.3) and parallel (Section 7.4.2.2) versions of the messy genetic algorithm. To allow a meaningful comparison, the parameter settings were set equal to the parameter settings used by Goldberg, Korb, and Deb in solving this problem.

7.2 Parameter Settings

The parameter settings used throughout the experiment are shown in Figure 53.

String Length	30
Block Size	3
Genic Alphabet	01
Reduction Rate	0.500000
Reduction Interval	3
Total Reductions	4
Shuffle Number	30
Cut Factor	0.016667
Splice Probability	1.000000
Maximum Generations	5

Figure 53. Messy Genetic Algorithm Parameter Settings

7.3 Sequential Implementation.

The sequential messy genetic algorithm was implemented then tested in accordance with the test strategy outlined in Chapter 5. The global optimal was found to a non-linear GA-hard problem very similar to used in the initial study on messy genetic algorithms (36:509-515). The same function was applied to the simple genetic algorithm. As expected, suboptimal results were returned. In what follows, the results obtained from parallelizing the messy genetic algorithm are presented.

7.4 Parallel Implementation.

7.4.1 Execution Times and Speedup. A fair amount of speedup was achieved by parallelizing the messy genetic algorithm. Figures 54 - 56 show the average run times for various phases of the messy genetic algorithms for 2 nodes, 4 nodes, and 8 nodes, respectively. The overall execution times decrease rather substantially as nodes are added. As expected, the initialization of the population showed a linear speedup. The results for the primordial phase were slightly surprising. In 2-node and 8-node configuration, slightly superlinear speedups were observed (Figures 59 and 60). Since the initial population was distributed equally among the nodes and everything else remained the same, a linear speedup was all that was hoped for. The reason for a superlinear was that while the shuffle number remained the same, a string did not, on average, have to search as long to find a compatible mate. Since the the construction of the building blocks and the juxtapositional phase were not parallelized, a fixed sequential time remains that is unaffected by the addition of processors. Extrapolation of the graph in Figure 57 would give a non-zero intercept. This fixed sequential component represents the lower bound of the run time. As Figure 58 indicates, adding processors quickly becomes a case of diminishing returns as the fixed sequential component begins to dominate.

7.4.2 Solution Quality.

7.4.2.1 Messy Genetic Algorithm. The parallel messy genetic algorithm consistently found the global optimal solution. Three different random number seeds were used in runs on a 2-node, 4-node, and 8-node cube. With the exception of one of the runs on the 2-node cube, every run returned the global optimal. It was very encouraging that every run on the 8-node cube returned the optimal solution, since this configuration had the minimum run time as well.

7.4.2.2 Simple Genetic Algorithm. The parallel simple genetic algorithm did not find the optimal solution for the non-linear problem. The best solution obtained was 296, and it was

found in generation 10 (out of 200 generations). The simple genetic algorithm typically found its best solution in an early generation, implying the solution was found more by chance than by directed evolution. The population tended to converge to a solution of 280 (a locally optimal solution) as the genetic algorithm proceeded, indicating the simple genetic algorithm was "deceived" by the non-linear function. Unless the simple genetic algorithm generates the optimum during random generation of the initial population or an early generation, it appears very unlikely the simple genetic algorithm can find the global optimum.

	Time (sec)	Percentage
Generate Competitive Template	0.013	0.0086
Create Building Blocks	0.002	0.0013
Initialize Population	8.446	5.563
Primordial Phase	123.531	81.37
Convert Data Structure	2.226	1.466
Juxtapositional Phase	17.093	11.259
Other	0.171	0.113
Total	151.814	1.000

Figure 54. Time Data for Various Parts of the mGA – 2 Nodes

	Time (sec)	Percentage
Generate Competitive Template	0.014	0.0161
Create Building Blocks	0.001	0.0011
Initialize Population	4.317	4.955
Primordial Phase	62.110	71.284
Convert Data Structure	2.815	3.231
Juxtapositional Phase	17.523	20.111
Other	0.303	0.348
Total	87.130	1.000

Figure 55. Time Data for Various Parts of the mGA – 4 Nodes

7.4.3 Comparison with Literature Results. In terms of solution quality, there was excellent agreement between the messy genetic algorithm created for this thesis effort and the messy genetic algorithm created by Deb, Goldberg, and Korb. Specifically, both returned optimal solutions for a "classical" GA-hard problem.

	Time (sec)	Percentage
Generate Competitive Template	0.013	0.0245
Create Building Blocks	0.001	0.0019
Initialize Population	2.262	4.255
Primordial Phase	30.503	57.374
Convert Data Structure	1.877	3.531
Juxtapositional Phase	17.783	32.320
Other	0.712	1.339
Total	53.165	1.000

Figure 56. Time Data for Various Parts of the mGA - 8 Nodes

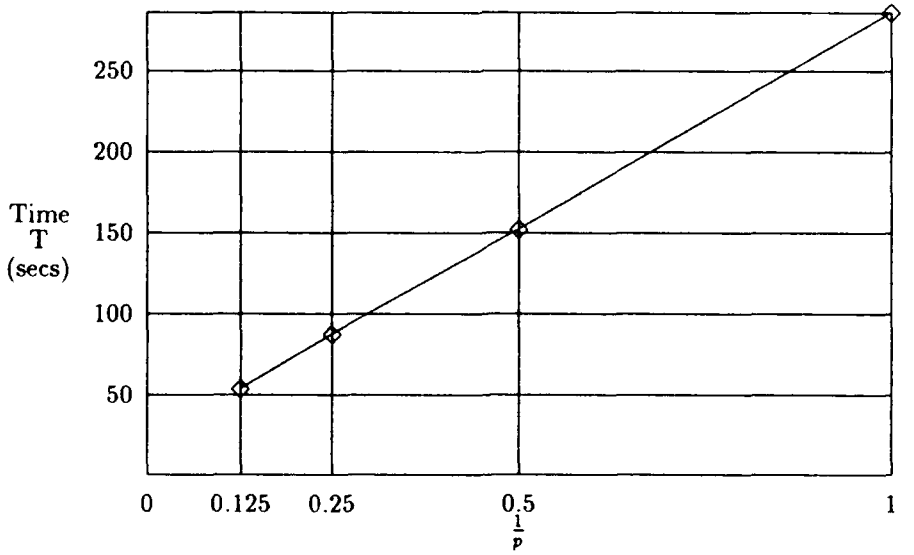


Figure 57. Overall Run Time versus Inverted Number of Nodes

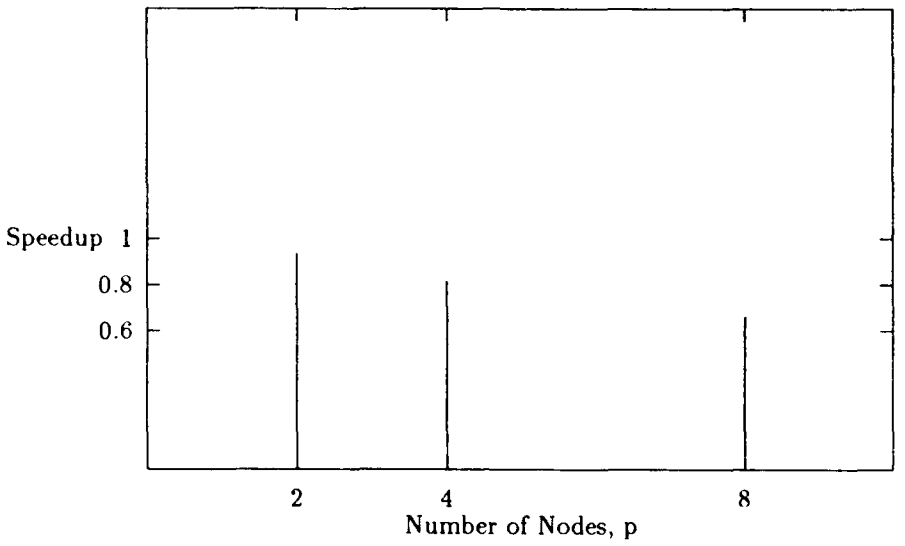


Figure 58. Speedup versus Number of Nodes

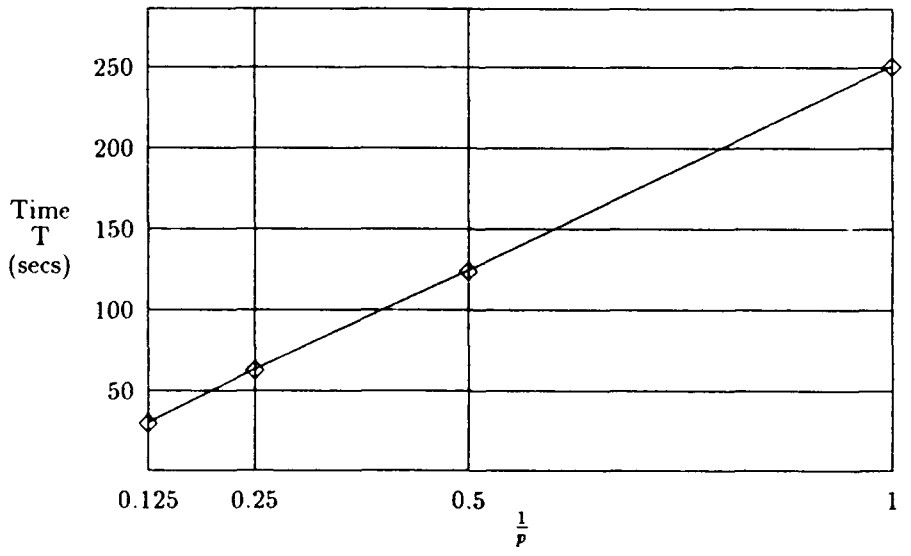


Figure 59. Primordial Phase Run Time versus Inverted Number of Nodes

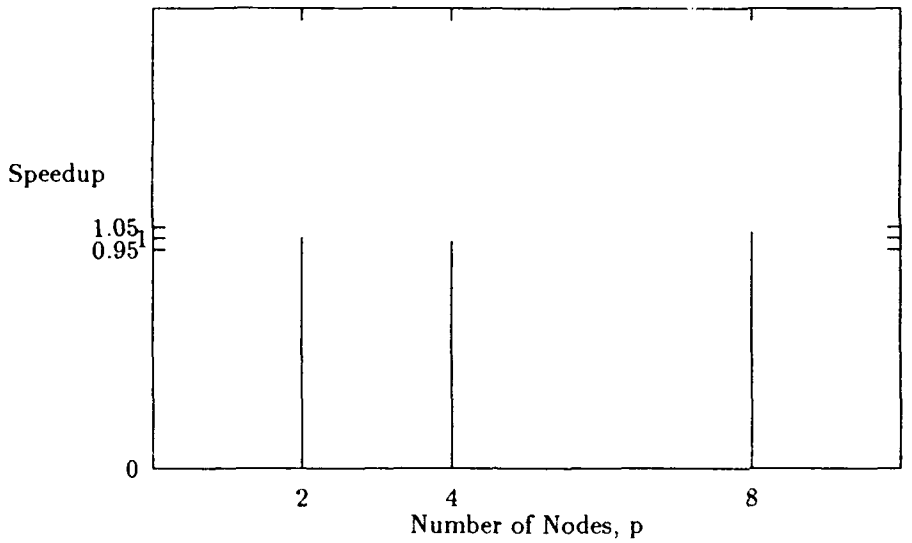


Figure 60. Primordial Phase Speedup versus Number of Nodes

Yet there were two fairly significant differences. The first difference was in the reduction interval required to give optimal solutions. For the non-linear test problem, the originators of the messy genetic algorithm report they were able to obtain the optimal solution even when they reduced the primordial population as often as every other generation. Reducing the population so often is certainly desirable since run time of the primordial phase is directly proportional to the population size. Yet when the same reduction interval was tried, the optimal result was rarely obtained. To consistently obtain the optimal result, the reduction interval had to be increased to three generations. The time lost by the greater reduction interval was offset in the juxtapositional phase, where the second major difference was noticed. The originators report 15 juxtapositional generations as the requirement for the optimal solution. In the implementation used in this thesis research, the optimal was found often in 4 generations and consistently in 5 generations.

The exact reason for these differences has yet to be determined. The results observed in AFIT's messy genetic algorithm do seem consistent with the implementation. Recall that tournament selection produces at most two copies of a solution in the new generation. Reduction is

performed by simply considering (by random selection) only half the population for a tournament. With only two solutions in the first generation, the best building block stands an excellent chance of not being selected for a tournament when reduction occurs in the second generation. With an increase in reduction interval, the best building blocks are in much higher concentration when a reduction actually occurs. Thus, elimination of the best building blocks is much less likely. The developers of the messy genetic information may have taken advantage of a priori knowledge of good building blocks to ensure these blocks were not eliminated when the population reduced, but this did not appear in their article.

The second difference is even more puzzling. The developers claim that 15 generations is the minimum requirement for a fully specified string of length 30 to be created from building blocks of length 3. Given that in the first generation two strings of length 3 could form 1 string of length 6 (no cut and a splice), then in the second generation two strings of length 6 could be spliced together to create a string of length 12, and so forth, it seems possible that a fully specified string could be created in 4 generations. If fact, a string could be as long as 48 ($= 3 * 2^4$). This hypothesis was confirmed by experimentation. The competitive template was set equal to 30 zeros and the maximum juxtapositional generations set equal to 4. An optimal solution consisting of 30 1's was returned, indicating the solution was constructed completely by cut and splice. Generalizing the calculation above and rearranging, gives the following equation for the minimum juxtapositional generations required for a fully specified solution of length l and building block size b:

$$gen_{full} = \lceil log_2(l/b) \rceil$$

Unless a user has heuristic information that allows the messy genetic algorithm to be terminated early, it is highly recommended that the maximum generations of the juxtapositional phase be set to at least gen_{full}. If this is done, it is possible for a string to form solely from building blocks, not simply part building block, part competitive template.

Such basic differences in the operation of the messy genetic algorithms point to different interpretations of the specifications. Additionally, there were several details not provided in the descriptions of the messy genetic algorithm (36) (37) (38). A case in point is the method for reducing the population mentioned previously. A dialogue with the developers has yet to resolve such discrepancies.

VIII. Conclusions.

The empirical and parametric nature of genetic algorithms makes it very difficult to generalize. All conclusions should really be qualified with "using these parameter settings", "against this problem", "with this communication strategy", and so forth. Any extrapolation, generalization, or induction based on the results observed in the experiments could lead to wrong conclusions. There is no basis to allow statements such as "increasing population size reduces premature convergence" or "messy genetic algorithms outperform simple genetic algorithms when applied to non-linear problems." The theory of genetic algorithms is not yet developed enough to allow such broad conclusions to be made. At best, the conclusions drawn from this research should only be used as guidelines, not as truisms.

8.1 Research Questions Conclusions.

1. Research Question #1 Conclusions. The standard Hypercube implementation of a genetic algorithm which shares solutions and uses local selection and mating is fundamentally different than a sequential genetic algorithm. The theoretical development of genetic algorithms assumes global selection and no restrictions on crossover, and nothing akin to sharing solutions. These differences do no necessarily mean that such Hypercube implementation is a poor search technique. In fact, the strategy using local selection and communication of solutions finds the optimal solution to the Rosenbrock's saddle on all 40 runs at a population size of 3200. For smaller population sizes, however, the algorithms (GN or PN) using global selection without sharing seemed more successful.

2. Research Question #2 Conclusions. Recall that the standard Hypercube implementation returns lower quality solutions than the sequential genetic algorithm when applied to Rosenbrock's saddle (73:156-158). Modifying the Hypercube genetic algorithm to implement global selection without sharing results in an algorithm that agrees better with Holland's theory.

Testing against Rosenbrock's saddle indicates the use of global selection can reduce the tendency toward premature convergence. As a result, for the majority of the population sizes examined, a global selection strategy outperforms the local selection strategy in terms of solution quality. So, in the range of populations sizes used in this study, the global selection strategy is more robust.

3. Research Question #3 Conclusions. In larger populations, sharing of the best solutions improves the likelihood of finding the global optimal. The communication of solutions amongst the nodes of a Hypercube seems to result in a synergism lacking in sequential genetic algorithms or parallel genetic algorithms that do not share solutions. However, with smaller populations, sharing of the best solutions seems to increase the tendency towards premature convergence. The population size where sharing changes from a liability to a benefit is very likely problem- and parameter-specific.

4. Research Question #4 Conclusions. In terms of solution quality, the results from the messy genetic algorithm developed during this research effort are in excellent agreement with the results of Deb, Goldberg, and Korb. In both cases, the messy genetic algorithm arrives at the optimal solution to a non-linear problem that a simple genetic algorithm cannot solve to optimality. From a more "white box" perspective, there seem to be differences. The AFIT messy genetic algorithm requires a longer reduction interval but fewer juxtapositional generations than the literature results. Differences in semantics might be the source of these differences. More important than these minor differences is the independent confirmation that messy genetic algorithms can solve GA-hard problems.

5. Research Question #5 Conclusions. Based on experimental results, it seems feasible to parallelize the messy genetic algorithm and still obtain the global optimal solution. A slightly better than linear speedup was obtained on the "bottleneck" of the genetic algorithm, the primordial phase. However, the juxtapositional does not seem amenable to parallelization.

Attempts to design the juxtapositional phase to operate in parallel were unsuccessful because the construction of the global optimal demands that the best building blocks reside on the same node. Certainly, since the building blocks in the target problem all had the same the same scale, a scheme could have been developed where each node sent its buildings blocks with highest fitness to all the other nodes. In general, however, the subfunctions do not have the same scale, so with most problems, such a scheme would fail. It seems the only way one can guarantee each node has all the best building blocks is to combine the post-primordial population from all the nodes. However, since by the end of the primordial phase the population is generally greatly reduced, combining populations is not a very expensive operation.

As the result of the juxtapositional phase and other components, there is a (virtually) constant sequential component to the execution times. The sequential components becomes increasingly significant as nodes are added to the cube. As a result, speedup decreases. Speedups range from 0.94 for 2 nodes to 0.66 for 8 nodes. With such diminishing returns, one cannot say that the implementation is scalable.

8.2 Summary.

This chapter offers conclusions to the research questions posed in Chapter 1. The goal to find strategies to reduce premature convergence in Rosenbrock's saddle seems to have been met. While the nature of genetic algorithms makes it hard to generalize, the fact that Rosenbrock's saddle is a difficult function for a genetic algorithm makes it possible that the strategies will be effective against many problems. The goal to implement and parallelize a messy genetic algorithm is a partial success. Results against a "classical" GA-hard generally corroborate the results of the developers in that the global optimal solution is consistently found. Yet while the parallel implementation offers the prospects for speedup without loss of optimality, there are limitations in terms of scalability. However, if the messy genetic algorithm proves ineffective against real-world problems, the efforts were for naught.

IX. Recommendations.

9.1 Introduction.

As little research has been directed towards genetic algorithms at AFIT, there are numerous possibilities for future genetic algorithm applications and research. Section 9.2 suggests problems to which genetic algorithms might be applied, while Section 9.3 details possible future areas of research.

9.2 Problem Recommendations.

9.2.1 Collaboration with Domain Experts. The list of genetic algorithm applications in Chapter 1 contains includes several application areas of interest to AFIT and the Air Force. As the tailoring of a genetic algorithm to a problem requires some domain-specific knowledge, a collaboration with the domain experts here at AFIT could be quite helpful in applying genetic algorithms to these application areas.

9.2.2 Application of Simple Genetic Algorithms to Non-Differentiable Functions. While optimal solutions were found to Rosenbrock's saddle, solution times were not impressive (even if the genetic algorithm had been terminated the moment the optimal solution was found). Other search techniques seem better suited. Since the function describing Rosenbrock's saddle is differentiable, a gradient-based technique may be used to find a solution (57). After tailoring of the gradient "step-size," a gradient-based search technique could arrive at a near optimal solution in roughly one second on a 386 personal computer (57). As shown by the performance curves shown Appendix A, a genetic algorithm takes much longer to obtain near-optimal solutions.

So while Rosenbrock's saddle was a good function on which to try premature convergence reduction techniques on, one would not generally use a simple genetic algorithm to solve a differentiable function if a gradient-based technique was available. For function optimization problems,

a genetic algorithm should be reserved for functions against which gradient-based techniques have difficulty. For instance, for functions having no analytical derivative, users of the gradient-based search strategies resort to Monte Carlo techniques requiring numerous iterations (57). The genetic algorithm, which does not require a derivative, is effectively immune to difficulties caused by non-differentiability. As a result, the genetic algorithm would likely be much more competitive in terms of time against non-differentiable function. One might perform a head-to-head competition pitting the genetic algorithm against other known optimizing strategies for non-differentiable functions.

An insight gained from this research is that a genetic algorithm will likely have difficultly competing against a search strategy tailored to a particular problem. Unlike the tailored search strategies, a genetic algorithm generally does not take advantage of problem-specific information that might help the search. The overall recommendation is to apply genetic algorithms where current search strategies perform poorly.

9.2.3 Application of Messy Genetic Algorithms to Real-Word Problems. While the messy genetic algorithm has had several initial successes, the test functions to which it has been applied seem to have little practical value (36) (37). The real test of the utility of messy genetic algorithms will come with their application against real-world problems. The developers believe "messy genetic algorithms now appear capable of solving many difficult combinatorial optimization problems to global optimality in polynomial time or better" (37:417) but have yet to demonstrate this. Such claims should be examined, especially against optimization problems of practical importance.

Application to non-linear problems seems especially appropriate given that messy genetic algorithms were designed for a class of non-linear problems (GA-hard). Sandgren cites several engineering design

optimization problems in his doctoral dissertation (70). As these design problems have varying degrees of non-linearity (70:27,29-31), the messy genetic algorithm might prove effective against them. A comparison could then be made between the results returned by the messy genetic algo-

rithm and the results returned by the suite of search methods used by Sandgren (70:28). The genetic algorithm system used by Forrest and Mayer-Kress against the same set of problems (25:312-331) might be included in the "competition" as well. The messy genetic algorithm might also be used to solve the non-linear equations used in international security models (65:312-333).

9.3 Recommendations for Future Research.

9.3.1 Termination Criteria/Solution Quality Indicator. A weakness in genetic algorithms is a lack of an adequate termination criteria. As was shown on the performance plots of the messy genetic algorithm, most runs became asymptotic long before 200 generations is reached. Methods have been devised to stop the genetic algorithm after the search seems to have converged. For example, the *Genesis* genetic program (41) has an option which, if selected, results in termination of the genetic program after a user-specified number of generations has passed without the generation of new solution. While termination criteria such as this could save time, the user is still given no indication as to the quality of the solution. One is left wondering whether the search converged onto a global optima or to an inferior local optima.

To get an idea of the quality of the solution found, some knowledge of the search space seems required. If one were to perform, say, a chi-square test for normality, some indication of the solution quality could be obtained. (Other distributions could be tested for as well). In particular, from calculations using the number of standard deviations that the fitness of the solution lies from the sample mean fitness, one could make statements such as, "The likelihood of selection of this solution from a random population is 0.001 percent." Such a measure would give a user a firmer indication of solution quality.

Use of standard deviations could provide a termination criteria as well. The user could specify up front that the search be terminated once a solution is found that is X standard deviations from the mean. Since no probability inference is being made, use of standard deviations in this

manner could be done whether or not the solution space is normally distributed. Requirements and suggestions for the design of a chi-square solution quality indicator are contained in the following paragraphs.

Generation of Sample Distribution. A statistical analysis of a population requires a sample distribution obtained by random sampling. If the initialization phase of the GA produces random solutions, the initial population may serve as the random sample. The sample mean fitness and standard deviation could then be calculated from the population members in the initial population. If heuristic information is used to create a better-than-average initial population, a module must be added to the genetic algorithm which takes a random sample (an unbiased estimate of the population average and variance is needed (12:23), but a heuristically generated population would be biased towards better-than-average solutions).

Sampling. If there are large number of feasible solutions to the problem, sampling with replacement can be used. In terms of the implementation of an algorithm, sampling with replacement means there is no need to check samples against previously generated samples and make changes if there is duplication. Not checking is justified if there is very little chance of duplication. For example, in a fully-connected N-city Traveling Salesman Problem, there are $(N-1)!/2$ solutions to the problem (5:1). Thus, for the problem sizes typically encountered in Traveling Salesman Problems, the chance of duplication of samples is very small. On the other hand, if there are relatively few solutions, as in a SCP with a relatively sparse matrix, checks for duplication must be performed (12:18).

The first question that needs to be answered is the size of the sample that needs to be taken. For "random sampling on a relatively large" population, a guideline for the sample size n is given by (58:368-369)

$$n = \frac{4\sigma^2}{25}$$

Since the population variance σ^2 is not known, it can be estimated from the variance of a previous random sample from the population (12:26,78).

Once this sample size estimate is obtained, a random sample of that size must be generated. The variance of this sample should then be compared to the original sample variance in terms of the estimate of the sample size. If the difference is great enough, a new random sample is generated based on the new estimate of the sample size (58:624).

Using the data obtained from the sample, a chi-square (χ^2) goodness-of-fit test should be performed "to determine whether a set of data could reasonably have originated from some given probability distribution" (54). This involves constructing a distribution table with intervals selected so that $n\hat{p}_i \geq 5$ for every interval i, where $n\hat{p}_i$ is the frequency expected for the probability distribution being considered, calculated using the maximum likelihood estimators for the unknown parameters (54:366). The hypothesis that the data is well described by the distribution function is accepted at the α level of significance if

$$\sum_{i=1}^{k} \frac{(x_i - n\hat{p}_i)^2}{n\hat{p}_i} \geq \chi^2_{1-\alpha,k-1-r}$$

where x_i are the observed frequencies and r is the number of parameters that were estimated (54:366-369). Say the data is well described by a normal distribution and the best solution is -5.98 standard deviations from the mean. One can then say the probability of randomly generating a solution better than the best solution is roughly 10^{-9} (9:412), giving at least some idea of the solution quality.

A chi-square test function was written and incorporated into the simple genetic algorithm. Unfortunately, Rosenbrock's saddle failed the test for normality. However, the use of a chi-square quality indicator is recommended for possible application to NP-Complete problems. Encouraging is the fact that the solution space of at least one NP-Complete problem (the Quadratic Assignment Problem) "tends to be normally distributed" (9:412).

9.3.2 Meta-Level Hypercube Implementation.

9.3.2.1 Background. A genetic algorithm has several parameters which must be specified. Among these are the population size, mutation rate, crossover rate, and selection strategy (40:124-125). The parameter instantiations define the adaptive strategy of the genetic algorithm. The performance of a genetic algorithm is a "non-linear function of these control parameters"(40:127). As genetic algorithms have been found to work well on complex functions, Grefenstette proposes a meta-level genetic algorithm, the upper level searching for optimal control parameters based on the performances of the lower level genetic algorithms (which are themselves searching for solutions to another optimization problem) (40:122-125). See Figure 61 (40:123).

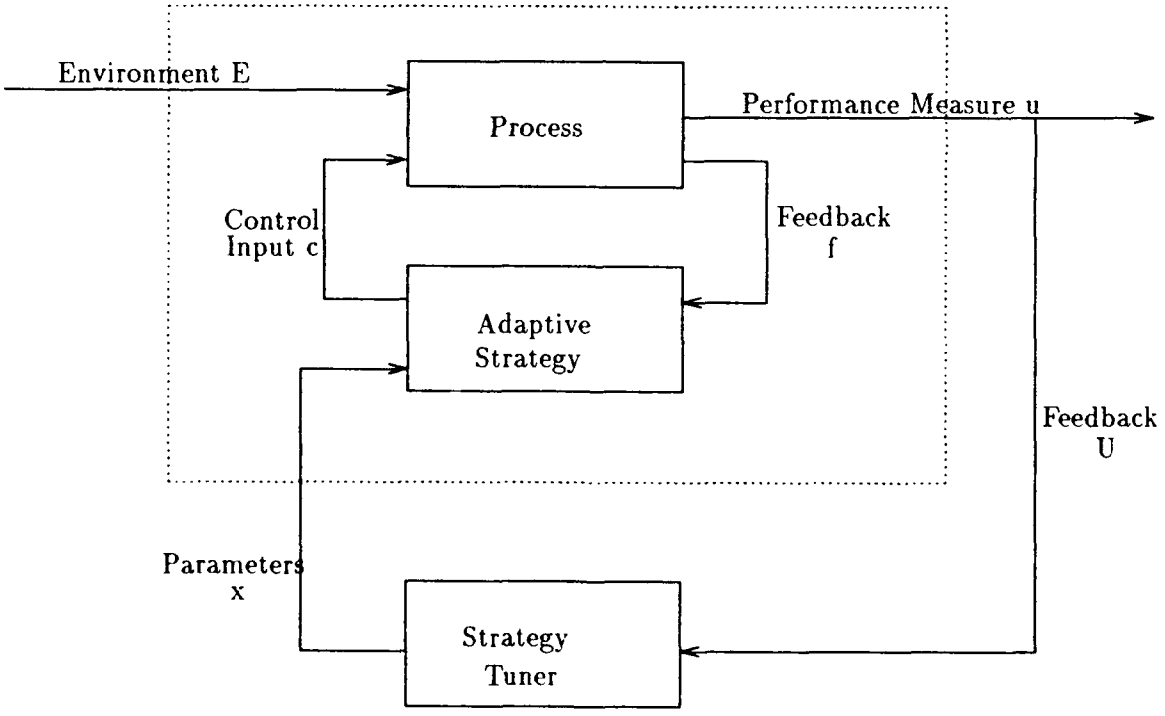


Figure 61. Grefenstette's Meta-Level Genetic Algorithm

What follows is an empirical discussion of Grefenstette's experimental procedure. He parameterizes the control value settings, creating 1000 instances of the genetic algorithm (GA). To create

an initial population, he evaluates each of the GA's against each of the five De Jong test functions (19). Randomly selecting 50 of these GA's and encoding them as "chromosomes", he then subjects them to the meta-level GA. The stepwise nature of the process suggests it is really not dynamic as Figure 61 seems to imply. Grefenstette does not give many details as to what is involved in the meta-level step, other than giving the control settings for the meta-level GA. Possibly the 2nd level GA changes the parameter settings of the 1st level GA's by subjecting them to selection, crossover, and mutation. After this meta-step completes (presumably after many repetitions), the 20 best GA's are subjected to more extensive testing. The overall best performer is compared to parameter settings suggested by De Jong in tests against an image registration problem. In terms online performance (overall average of solutions), a GA using parameter settings determined by the meta-level genetic algorithm outperform a GA using De Jong's parameters. In terms of offline performance (average of the best solutions), the GA's are statistically identical (40:127-128).

Grefenstette notes the following problems with his meta-level process:

- Good performance can be obtained with a variety of control parameters, anyway.
- The experiments required a great deal of CPU time.

Grefenstette suggests a meta-level GA that would allow the dynamic variation of control parameters, but questions the feasibility since "for many optimization problems, the number of evaluations which can be performed in a reasonable amount of time would not allow the GA enough evaluations to modify its search techniques to any significant degree" (40:128).

Based on the inefficiency associated with the meta-level GA, its implementation seems impractical. That is, until one recalls that Grefenstette performed his experiments on a sequential computer. Since evaluations can be performed in full parallel, use of a Hypercube could alleviate the time problem Grefenstette mentions. Also, as discussed next, use of a Hypercube would allow the implementation of a truly dynamic meta-level GA.

9.3.2.2 Suggested Meta-Level Genetic Algorithm Hypercube Design. In the current Hypercube implementations, such as the one done by Pettey, the nodes of the Hypercube each run a GA with the same control settings and share best solutions. Another approach might be to assign different control settings to each node, then share the settings that appear to work to best. However, this does not allow for variation from the original n sets of control strategies. So a better approach might be to have a supervisor running a 2nd level GA that would allow the control parameters to evolve.

A meta-level GA was implemented on AFIT's iPSC/2 Hypercube, but time limitations prevented further study. A suggested plan-of-attack for future research is as follows:

- Port the current meta-level GA on the iPSC/2 to the iPSC/1 (or another Hypercube with more nodes). This would allow a population of 32 parameter sets (the 8 nodes of the iPSC/2 might be too small a population).
- Begin by keeping all parameters but mutation rate constant, and study how the mutation rate evolves. (The current implementation is implemented so as to allow only the mutation rate to change). Compare the solutions returned by the meta-level GA with solutions returned with typical static mutation rate settings (usually low but non-zero settings work best (40:127)).
- If results from this the "proof of concept" are encouraging, extend the meta-level GA to the other GA parameters.

9.4 Summary.

This chapter offers suggestions for future genetic algorithm applications (Section 9.2) and research (Section 9.3). Several other possibilities exist, ideas for which may be obtained from the literature (see Bibliography).

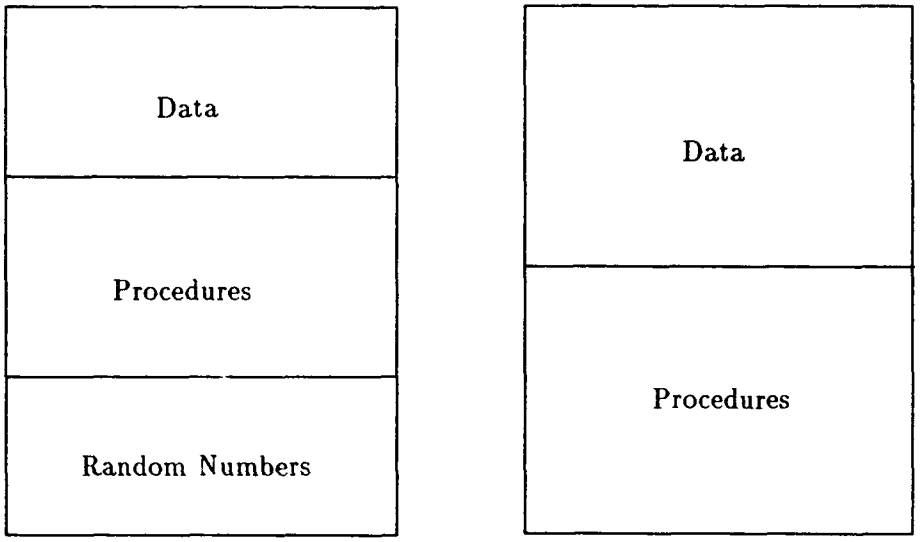
Appendix A. *Parallel Random Number Generation.*

A.1 Requirements.

While the subject of random number generation for genetic algorithms does not receive much attention in the literature, the implementation of the stochastic features of genetic algorithms seems to rely very heavily on random number generators (34:62-70) (41). For example, Grefenstette's *Genesis* code uses a a random number generator to initialize the populations, to randomly select parents for mating (done implicitly by randomly shuffling the population), randomly selecting string positions to be mutated, and so forth. In parallel genetic algorithm, random number generators could be involved in randomly selecting individual(s) to be communicated and in selecting individuals to be replaced by the individuals received by other nodes, in addition to all the operators mentioned above (63:156) (49:171) (73:178-179).

Since a genetic algorithm is modeling the probabilistic processes involved in evolution, the fidelity of the model would seem to rely rather heavily on the pseudo-random number generator. This fact remains true whether the genetic algorithm is implemented on either a sequential or parallel machine. With this in mind, it seemed incredible that no study could be located that examines the effect of the the type of random number generator used, especially since many common methods of random number generation are unsatisfactory because the sequences generated tend to repeat (52:3,4). The lack of a study may be due to the fact that a genetic algorithm will function regardless of whether the random number generator is good or not. For example, a genetic algorithm would still return an answer even if the "random number generator" simply alternated between a 1 and a 0 every time it was called.

Stochastic programs are defined not only by the algorithm and the state of the data being operated on, but also the random number sequence (Figure 62). A stochastic algorithm cannot operate on the data until it becomes deterministic through the use of a random number (let's call this process "instantiating" the stochastic algorithm). "Instantiating" a stochastic algorithm



(a) Probabilistic Algorithm (b) Deterministic Algorithm

Figure 62. Random Numbers are an Integral Part of a Stochastic Algorithm

with different random number sequences creates different deterministic algorithms. Notice that while the a random number is typically not generated until it is needed, a stochastic function could pre-generate its random number sequence (provided that no conditional statements make this impossible).

For a parallel implementation of a genetic algorithm, one must also consider that random number generators used in computer implementations are pseudo-random. A pseudo-random number generator generates a sequence of numbers that appear to be random. However, the numbers are not truly random since a each number (except the first) is calculated from the previous number(s) in the sequence (52:3). Given the same starting number ("the seed") the same sequence of random numbers is produced. This deterministic method of generating random numbers has "worked quite well in nearly every application (52:3)," and the repeatability is quite often quite desirable. Indeed, debugging a program or studying the effects of a modification would be more difficult if the random number generator used was truly random (27:202).

However, the use of pseudo-random number generators offers the potential for "correlations among the sequences" of random numbers on each processor of a parallel computer. While Fox does not give a precise definition of "correlation," from the context of his discussion, he seems to mean "related." Such "correlations" can result in "several severe problems" including the reduction in efficiency due to redundancy and "incorrect results" due to bias (27:202). For a search algorithm, redundancy causing an inefficient search would seem to be the more likely problem. A "incorrect result" does not seem possible because solutions are normally checked for feasibility.

However, it is hard to imagine a scenario (other than the Monte Carlo algorithm (27:202)) in which such problems would arise. Control decompositions seem relativity immune to the correlation problem. In control decomposition, each processor will be executing a different program (either stochastic or deterministic). So even if each processor generated the exact same random number sequence (perfect "correlation"), the efficiency would not be diminished because the random numbers are applied differently. In a data decomposition, each processor is assigned different data. Identical random number sequences result in identical programs, but each act on different data. If the programs, data, and random number sequences are identical, however, the effect is complete redundancy on the processors. See Figure 63 for a summary.

A.2 Examination of the Parallel Random Number Generator.

With the extensive role a random number generator plays in a genetic algorithm, a thorough examination of the random number generator used in Sawyer's parallel version of Grefenstette's code, one of the foundations of this thesis effort, seems warranted.

Sawyer parallelized Grefenstette's *Genesis* program for his term project in the Parallel Algorithms course here at AFIT (71:). This program was to be the foundation for this research effort. In light of the importance and potential for difficulties associated with the random number generator

Data	*Algorithm*	*Random Number* Sequence*	*Problem*	*Comment*
Different	Different	Different	No	Control and Data Decomposition
Different	Different	Same	No	← Assumes no bias in the random number sequence
Different	Same	Different	No	What Sawyer tried to do, but only assured not identical—still might be correlated
Different	Same	Same	No	Once "instantiate" stochastic function with seed, analogous to data decomposition of a deterministic algorithm
Same	Different	Different	No	Control Decomposition—different random number sequences unnecessary since using different stochastic functions
Same	Different	Same	No	Effective control decomposition—identical random number sequences applied to different functions
Same	Same	Different	No	Still an effective control decomposition since identical stochastic functions "instantiated" with different random numbers are different deterministic function
Same	Same	Same	Yes	Processors perform same function on same data; amount of loss of effective speedup proportional to amount of correlation

* For random number sequences, define different to mean uncorrelated and same to be either identical or correlated.

Figure 63. Possible Stochastic Algorithm Relationships on the Processors of a Parallel Computer

for a parallel genetic algorithm, the method of generating random numbers used by Grefenstette and Sawyer was examined.

Grefenstette defined the random number generator as follows:

```
#define Rand()  (( Seed = ( (Seed * PRIME) & MASK) ) * SCALE )

#define Randint(low,high) ( (int) (low + (high-low+1) * Rand()))
```

where, according to the comments in the code, Rand() returns a "pseudo-random value between 0 and 1, excluding 1, and Randint returns a pseudo-random number in the interval [low, high). Seed is a variable of type int, the first value of which is input by the user. PRIME, MASK, and SCALE are constants having the values shown in Figure 64.

Parameter	Definition in Code	Description
PRIME	65539	A prime number
MASK	$(0 << (INTSIZE - 1))$	Masks high order bit
INTSIZE	Number of bits in integer	System Specific
SCALE	0.4656612875e-9	Reciprocal of 7fffffffh (Makes range [0..1.0])

Figure 64. Parameters in Random Number Generator in *Genesis*

In Sawyer's parallel version, the user input seed, renamed OrigSeed, is used to calculate a starting nodal seed as follows:

```
OrigSeed = (unsigned) ((OrigSeed + My_node) / (My_node + 1));

Seed = OrigSeed;
```

Neither Grefenstette's nor Sawyer's code or associated documentation discuss how the random generator works, defend its choice over other methods of random generation, or even give a reference from whence it came. Additionally, while one suspects Sawyer performed the preceding operation to prevent nodes from having the same starting seed (that is, to prevent perfect "correlation"),

again this is discussed nowhere. Given the criticality of the random number generator, these issues must be addressed.

A.3 Does the Sequence Appear to be Random?

Figure 65 shows 30 10-digit numbers generated by the pseudo-random number using the program shown in Figure 68. Figure 66 shows the groupings of zeros for the Chi square test.

```
1011111000
1001100010
1010111000
1001000110
0100011000
1110010010
1000111001
0111101000
1100100010
0000100011
1001100000
1101011000
1110010101
0000111110
0001000011
0100101000
1010111010
1010100100
0110100010
1100011000
1000111111
0101011111
0011101111
1010101000
0111111001
0001100001
1100110110
0110110000
1111101110
1110011001
```

Figure 65. Pseudo-Random Number Generator Test Data

Following the example in Conover's book, the blocks j = 0,1,2,3 and j=7,8,9,10 are combined as shown in Figure 67.

```
          (Number of Blocks of 10 containing j zeros)
j   0   1   2   3   4   5   6   7   8   9   10   Total
    0   0   1   4   5   6   8   6   0   0   0    30

                    154 zeros
                    146 ones
```

Figure 66. Distribution of Zeros

Class	$j \leq 3$	$j = 4$	$j = 5$	$j = 6$	$j \geq 7$	Total
O_j = *observed number*	5	5	6	8	6	30
E_j = *expected number*	5.157	6.153	7.380	6.153	5.157	30

Figure 67. Data Bins for the Chi-Square Test

The test statistic is (15:187)

$$T = \sum_{j=1}^{c} \frac{(O_j - E_j)^2}{E_j}$$

Substituting the experimental values gives

$$T = \frac{(5 - 5.157)^2}{5.157} + \frac{(5 - 6.153)^2}{6.153} + \frac{(6 - 7.380)^2}{7.380} + \frac{(8 - 6.153)^2}{6.153} + \frac{(5.157 - 6)^2}{5.157}$$

$$T = 1.17112$$

For $(c - 1) = (5 - 1) = 4$ degrees of freedom, where c is the number of "bins" into which the data was grouped,

Since the data was group into 5 bins and no parameters were estimated, the chi-square random variable has $(5 - 1) = 4$ degrees of freedom (15:190). The observed value of T is very small, much smaller than even the 0.75 quantile of a chi-square random variable with four degrees of freedom, 5.385 (15:367). Thus, the 1's and 0's appear to be randomly generated.

```
#include <stdio.h>
#define PRIME 65539
#define SCALE 0.4656612875e-9
#define INTSIZE 32
#define MASK ~(~0<<(INTSIZE-1))
#define Rand()  (( Seed = ( (Seed * PRIME) & MASK) ) * SCALE )
#define Randint(low,high) ( (int) (low + (high-low+1) * Rand()))
main()
{
  int i,j;                /* Loop Counters */
  int even = 0;           /* Number of even (zeros) within a group */
  int num[11];            /* The Groupings for the Chi-square test */
  int Seed = 123456789;   /* Seed for random number generator */
  int total = 0;          /* Total number of groupings */
  for (i=0; i<11;i++)     /* Initialize groupings */
    num[i] = 0;
  for (i=1; i<301;i++)    /* Generate 300 1's and 0's using pseudo-random */
  {                       /* number generator, counting the number of     */
  j = Randint(0,1);       /* of zeros in each group of 10 digits          */
  printf("%d",j);
  if (j == 0)
    even++;
  if (i % 10 == 0)        /* Check whether at end of a group of 10 */
    {
    num[even]++;          /* Increment count of the group containing */
    even = 0;             /* even 0's, then reset the counter even  */
    printf("\n");
    }
  }
  printf("\n");            /* Print out the number of observations of */
  for (i=0; i < 11; i++)  /* each group, followed by the Total number */
     printf("%d   ",i);   /* of instances (should be 300/10=30) */
  printf("Total\n");
  for (i=0; i < 11; i++)
     {
     printf("%d   ",num[i]);
     total += num[i];
     }
  printf("   %d\n",total);
}
```

Figure 68. Program to Generate Data for Chi-Square Test for Randomness

A.4 Prevention of Perfect "Correlation."

In the early tests of the baseline genetic algorithm, small random number seeds were used. The results returned were quite poor, but they were consistently poor—every node returned the exact same answer. Following the advice of Sawyer[1], much larger seeds were used. Solution quality increased immensely, and often each node returned a different answer.

The reason for the poor results was that Sawyer's nodal seed calculation does not prevent nodes from starting with the same pseudo-random number generator seed. This is illustrated in Figure 4 which plots the starting nodal seeds versus user input seeds for nodes 6 and 7. The integer division involved in the calculation of the nodal seeds results in a step function. For many low input seeds, there is significant overlap in the "plateaus," indicating identical nodal seeds. Starting with identical nodal seeds is disastrous. The first use of the random number generator is in the generation of the nodal population. So starting with identical seeds results in identical nodal population. Since identical programs are running on the nodes, the search performed is completely redundant. So the occurrence of identical nodal seeds had to be prevented.

The overlap in the "plateaus" is observed to diminish as the user input seed increases. The overlap disappears completely for user input seeds greater than or equal to $N^2 - 2N + 2$. Using this fact, Figure 70 lists the minimum user input seeds needed to prevent any of the nodes from starting with the same random number seeds.

A.5 Ramifications of Overlap.

While the guidance given above will prevent the nodes from starting with the same seed, there does not seem to be anything preventing the "correlation" such as that shown in Figure 71. Pictured are the random number sequences of two nodes. Should the nth random number of the first sequence (A_0) be equal to the seed of the the second random number sequence (B_0), the two

[1] Personal interview

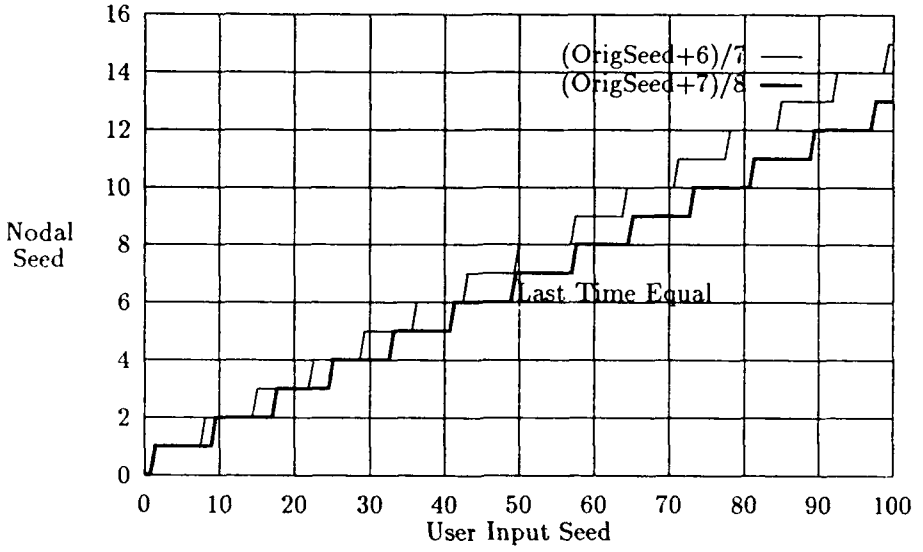


Figure 69. Nodal Seeds versus User Input Seeds for Nodes 6 and 7

Number of Nodes	*Minimum Seed*
1	1
2	2
4	10
8	50
16	226
32	962
64	3970
128	16130

Figure 70. Minimum User Input Seeds for Random Number Generator

sequences will be related as follows:

$$A_{i+n} = B_i$$

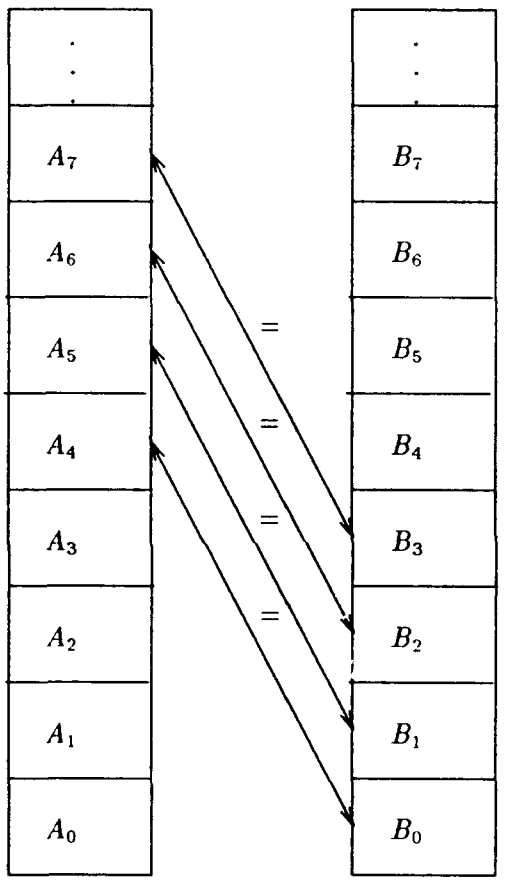


Figure 71. Correlation Between Nodal Random Number Sequences

For a parallel genetic algorithm, if such a correlation would occur in the early stages of generation of the initial populations, several of the starting on the two nodes could be identical. Specifically, if n be a the start of a string (that is, (n mod string_length) = 0), the strings generated on the nodes will have a relationship similar to that shown in Figure 1. Intuitively, it would seem having several identical strings at the start of the genetic algorithm might limit the breadth of the search. Certainly, having identical strings would cause the parallel version to diverge for a sequential

version, since the generation of identical strings in the starting population of a sequential genetic algorithm is extremely unlikely.

While the use of the prime number in the generator and Sawyer's massaging of the seed may prevent such a correlation form occurring, this could not be proven. Fox does give a method to ensure a "correlation" as shown in Figure 71 never occurs (27:203-207). This is discussed further next.

A.6 A Final Misgiving with the Random Number Generator.

As stated previously, it would be desirable if the parallel genetic algorithm would perform as close as possible to the sequential version, only faster. In this way, the heuristic knowledge (optimal crossover rates, mutation rates, and so forth) available from years of study with sequential genetic algorithms could be directly used.

The current method of random number generation causes the parallel and sequential versions to differ in function as well as speed. With Sawyer's seed massaging, the resulting random number sequences on the parallel processors, taken as a whole, are not equivalent to the sequential version. Fox offers a method of allowing this equivalence by staggering the random number sequences on each processor of the parallel computer (27:202-205). For example, consider a 2 processor parallel computer The first processor generates the 1st, 3rd, 5th, ...random numbers of the sequential random number sequence, while the second processor generates the 2nd, 4th, 6th, ... However, time limitations prevented the implementation of Fox's algorithm. This is not believed to be a serious flaw, and this difference might contribute to the superior answers often found with parallel genetic algorithms. That is, perhaps the difference parallel random number sequences from the sequential sequence allows more extensive searching and avoids premature convergence.

Appendix B. *Experimental Data–Examination of Premature Convergence using Rosenbrock's Saddle*

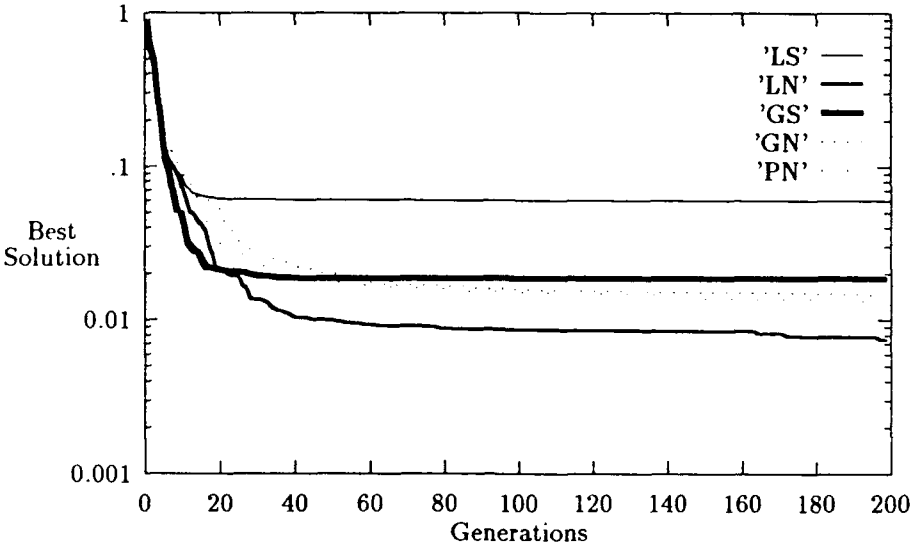


Figure 72. Best Solution Evolution – Population Size 80

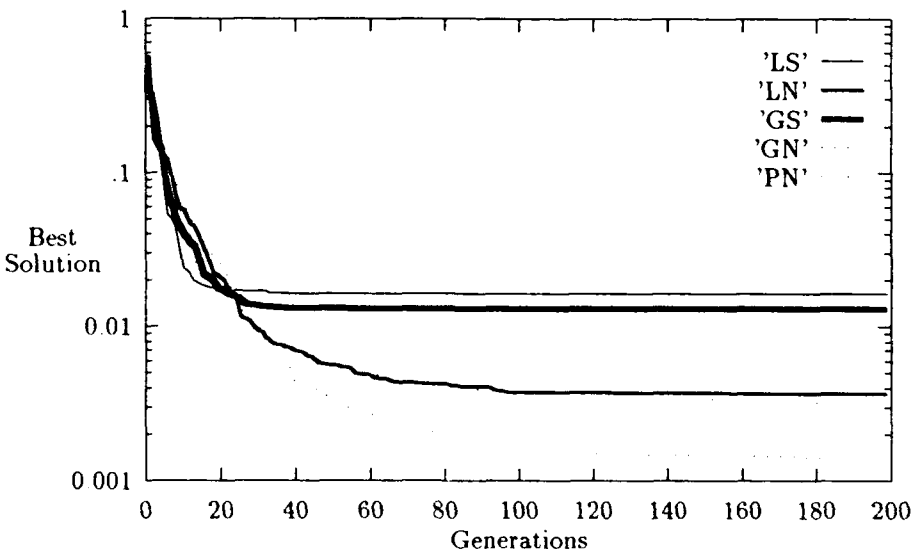


Figure 73. Best Solution Evolution – Population Size 120

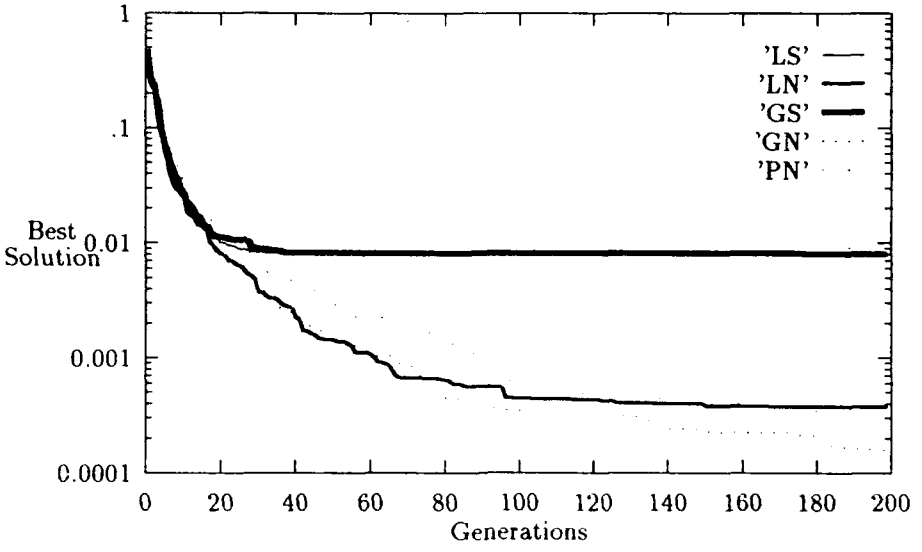


Figure 74. Best Solution Evolution - Population Size 160

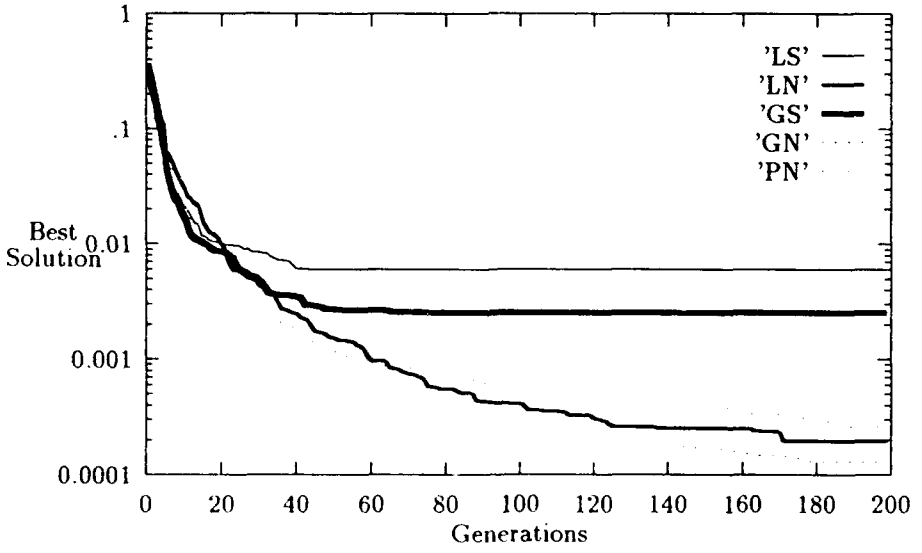


Figure 75. Best Solution Evolution - Population Size 200

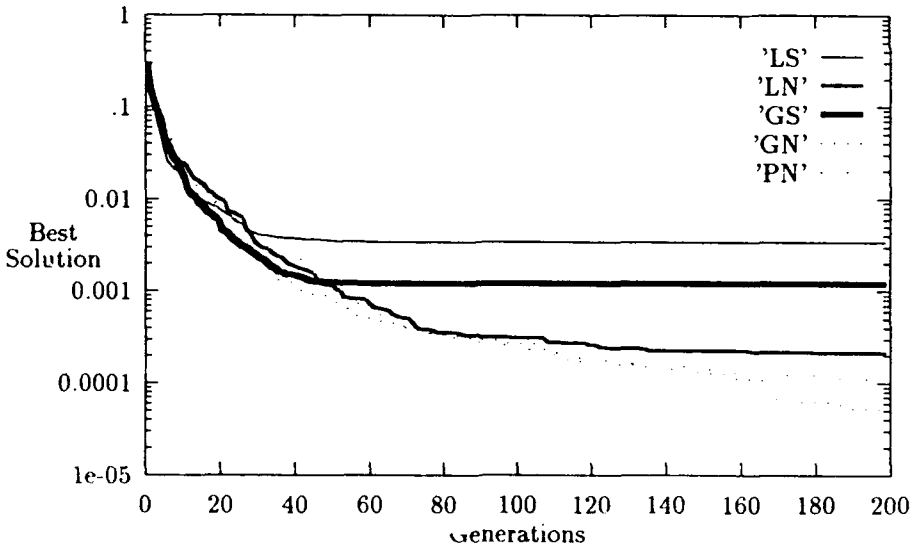


Figure 76. Best Solution Evolution – Population Size 240

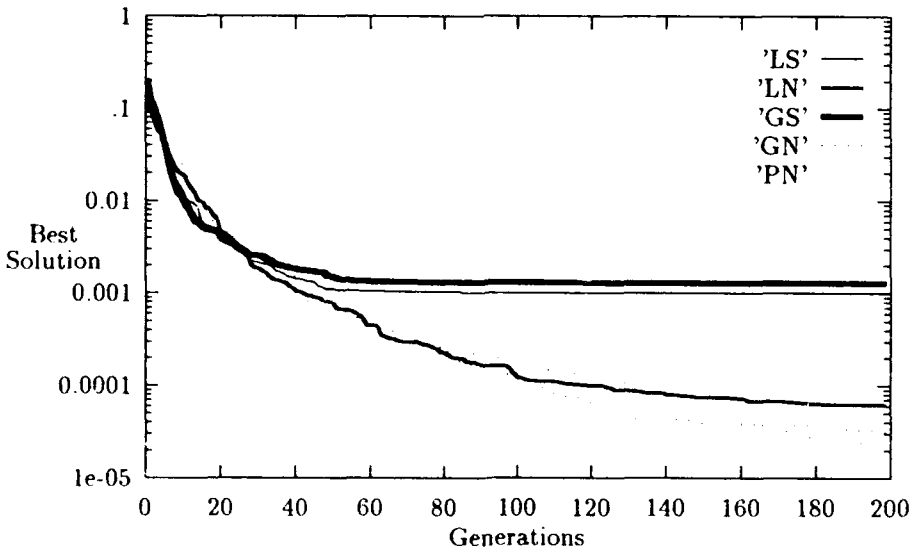


Figure 77. Best Solution Evolution – Population Size 280

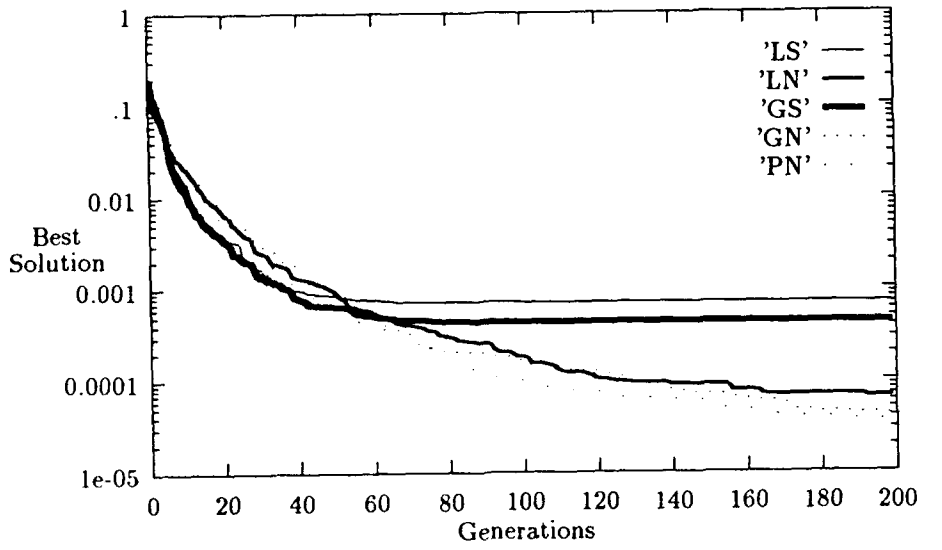


Figure 78. Best Solution Evolution - Population Size 320

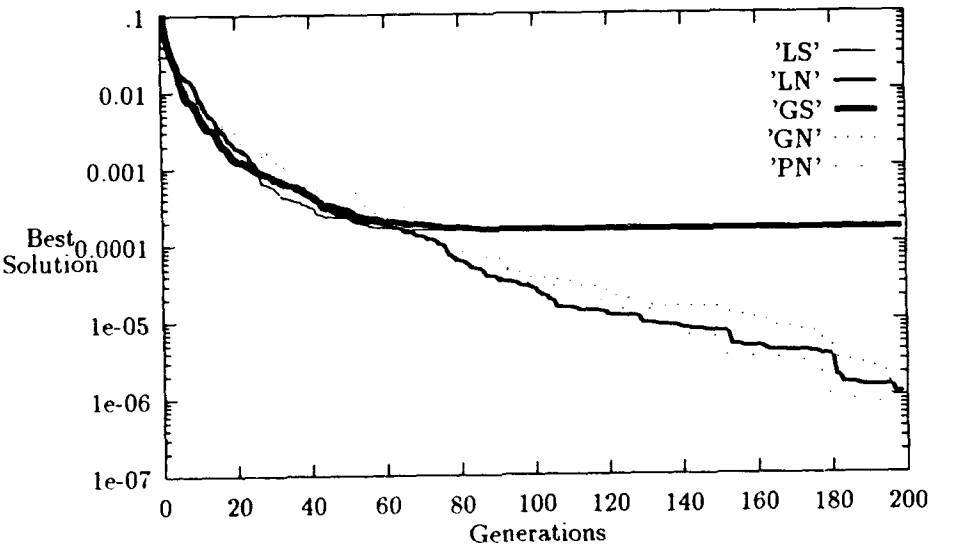


Figure 79. Best Solution Evolution - Population Size 640

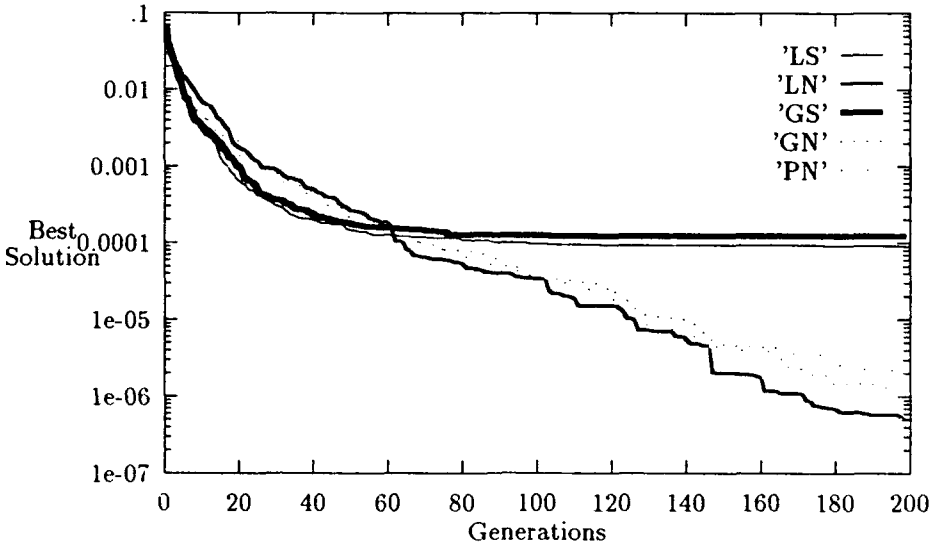


Figure 80. Best Solution Evolution – Population Size 960

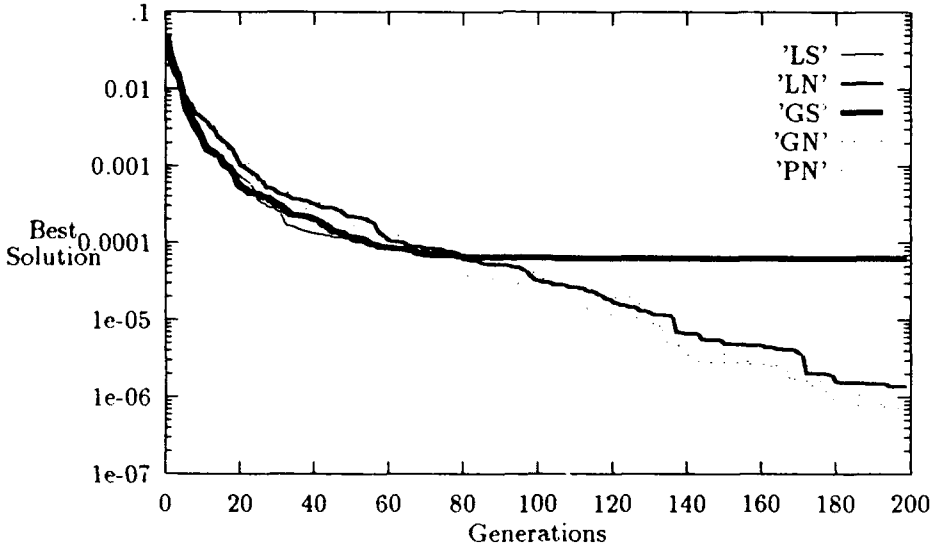


Figure 81. Best Solution Evolution – Population Size 1280

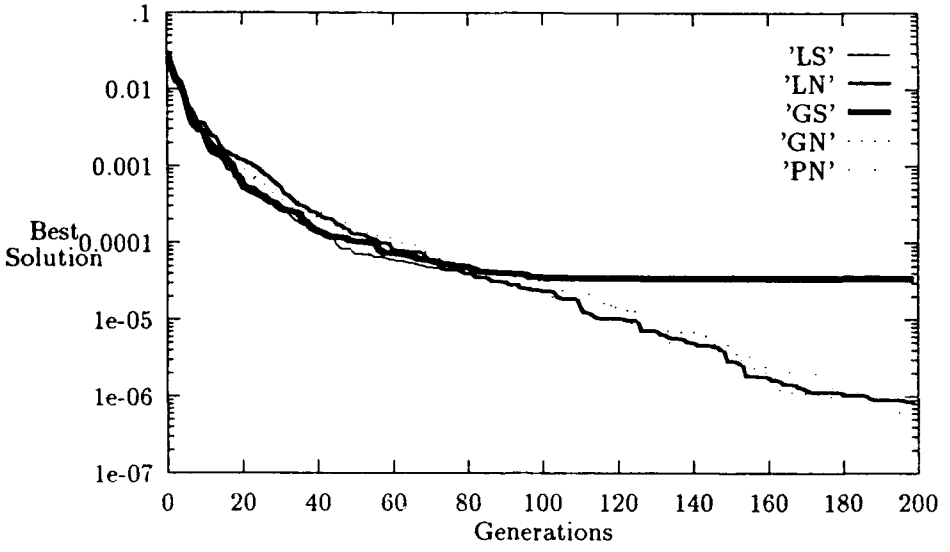


Figure 82. Best Solution Evolution – Population Size 1600

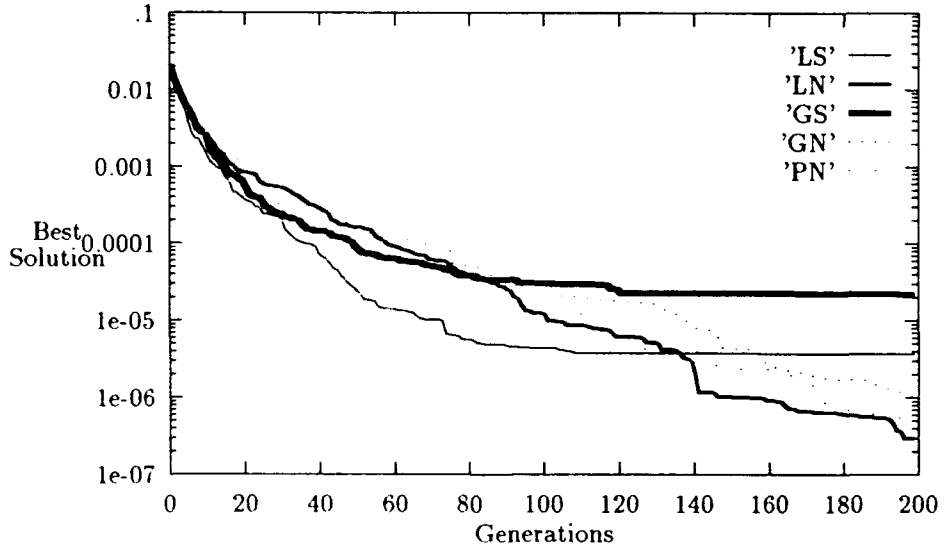


Figure 83. Best Solution Evolution – Population Size 1920

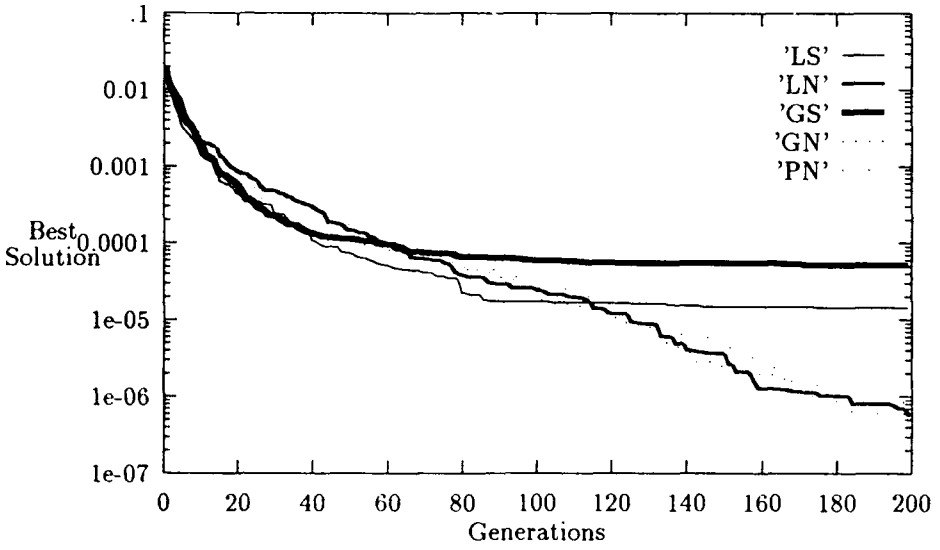


Figure 84. Best Solution Evolution – Population Size 2240

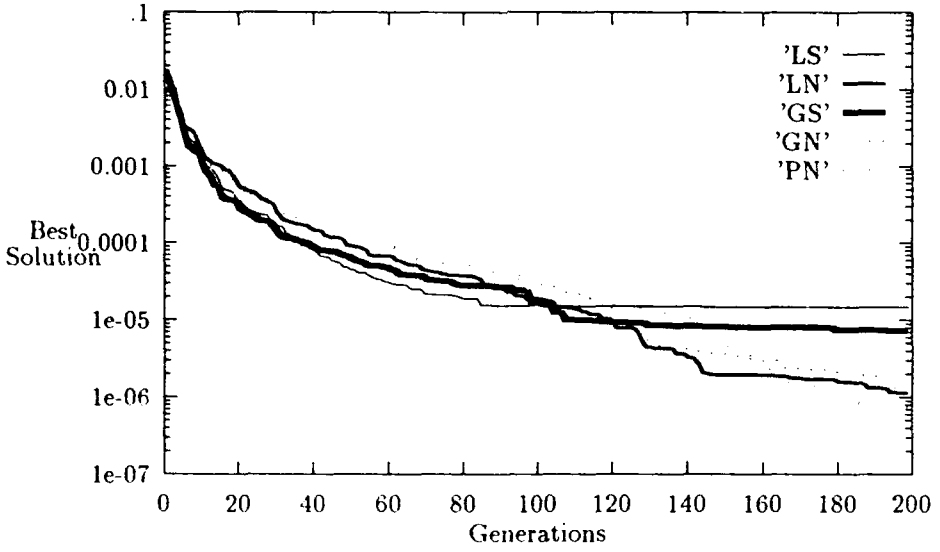


Figure 85. Best Solution Evolution – Population Size 2560

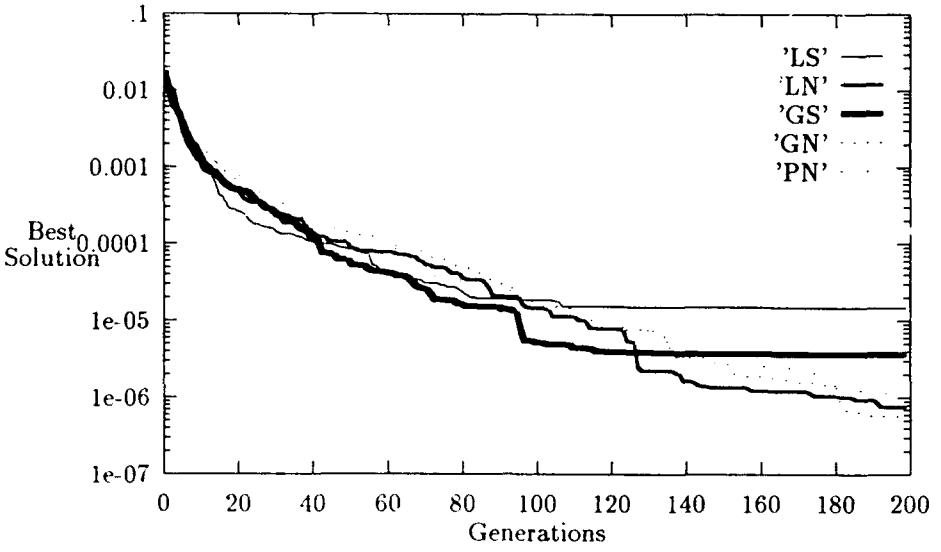


Figure 86. Best Solution Evolution – Population Size 2880

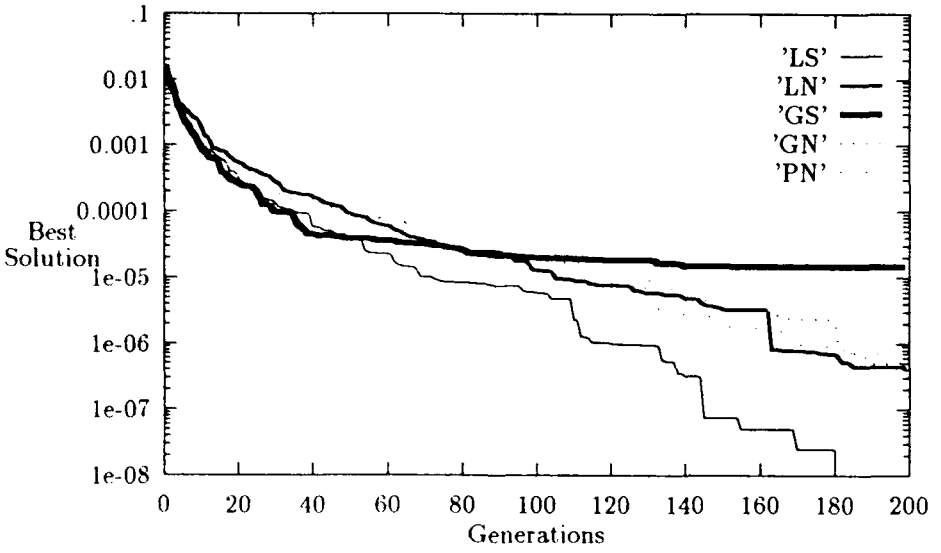


Figure 87. Best Solution Evolution – Population Size 3200

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0600974961	0.0073666520	0.0185025357	0.0146298287	0.0132491450
Variance	0.0073413934	0.0001211785	0.0005285001	0.0005358038	0.0007064640
Standard Deviation	0.0856819315	0.0110081119	0.0229891306	0.0231474354	0.0265793912
Best = Optimal	0	0	0	1	0
Percentage	0.000	0.000	0.000	2.500	0.000
Best Generation					
Average	35.950	84.250	39.625	122.025	106.575
Variance	959.997	3001.731	1439.779	2502.743	3327.122
Standard Deviation	30.984	54.788	37.944	50.027	57.681
Run Time (sec)					
Average	1.412	1.259	4.305	5.437	3.078
Variance	0.000	0.000	0.019	1.495	0.207
Standard Deviation	0.003	0.004	0.137	1.223	0.455

Figure 88. Performance Statistics - Population Size 80

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0163581096	0.0036904968	0.0131689681	0.0014203140	0.0032367282
Variance	0.0006574382	0.0000537782	0.0003680278	0.0000137595	0.0000936323
Standard Deviation	0.0256405569	0.0073333611	0.0191840510	0.0037093797	0.0096763761
Best = Optimal	0	0	0	2	1
Percentage	0.000	0.000	0.000	5.000	2.500
Best Generation					
Average	28.600	83.275	28.875	109.050	105.675
Variance	216.144	1783.640	368.471	1975.074	1795.917
Standard Deviation	14.702	42.233	19.196	44.442	42.378
Run Time (sec)					
Average	2.014	1.856	5.496	5.985	3.627
Variance	0.000	0.000	0.036	0.317	0.193
Standard Deviation	0.003	0.004	0.190	0.563	0.440

Figure 89. Performance Statistics - Population Size 120

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0081064174	0.0003744654	0.0080339205	0.0001592092	0.0003291361
Variance	0.0002224297	0.0000005102	0.0001655790	0.0000000970	0.0000004910
Standard Deviation	0.0149140776	0.0007142931	0.0128677485	0.0003114735	0.0007007261
Best = Optimal	1	5	1	2	3
Percentage	2.500	12.500	2.500	5.000	7.500
Best Generation					
Average	30.425	89.200	36.750	127.775	122.275
Variance	211.430	1409.190	785.218	3124.948	1827.025
Standard Deviation	14.541	37.539	28.022	55.901	42.744
Run Time (sec)					
Average	2.640	2.482	6.682	7.204	4.357
Variance	0.000	0.000	0.060	0.380	0.144
Standard Deviation	0.005	0.004	0.244	0.617	0.379

Figure 90. Performance Statistics – Population Size 160

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0059918432	0.0001961058	0.0025524294	0.0001286657	0.0002618076
Variance	0.0002409487	0.0000001538	0.0000147185	0.0000000665	0.0000002075
Standard Deviation	0.0155225211	0.0003921655	0.0038364707	0.0002578474	0.0004554858
Best = Optimal	2	5	1	7	3
Percentage	5.000	12.500	2.500	17.500	7.500
Best Generation					
Average	35.300	102.425	41.575	126.500	135.575
Variance	191.600	1770.558	385.430	2426.410	1815.430
Standard Deviation	13.842	42.078	19.632	49.259	42.608
Run Time (sec)					
Average	3.244	3.079	7.820	8.638	5.610
Variance	0.000	0.000	0.073	0.711	0.332
Standard Deviation	0.010	0.004	0.271	0.843	0.577

Figure 91. Performance Statistics – Population Size 200

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0033921070	0.0002126956	0.0011919321	0.0000507796	0.0000979831
Variance	0.0001037008	0.0000001445	0.0000089784	0.0000000119	0.0000000344
Standard Deviation	0.0101833591	0.0003801375	0.0029964003	0.0001091300	0.0001854318
Best = Optimal	3	7	3	7	11
Percentage	7.500	17.500	7.500	17.500	27.500
Best Generation					
Average	44.775	108.150	44.300	152.375	140.450
Variance	755.307	1750.951	281.395	954.804	2123.126
Standard Deviation	27.483	41.844	16.775	30.900	46.077
Run Time (sec)					
Average	3.872	3.704	8.989	10.048	6.587
Variance	0.000	0.000	0.129	0.928	0.442
Standard Deviation	0.010	0.004	0.359	0.964	0.665

Figure 92. Performance Statistics – Population Size 240

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0010071887	0.0000631100	0.0013087534	0.0000313350	0.0000240557
Variance	0.0000083784	0.0000000243	0.0000085081	0.0000000095	0.0000000090
Standard Deviation	0.0028945536	0.0001558943	0.0029168696	0.0000976677	0.0000950773
Best = Optimal	4	11	1	15	13
Percentage	10.000	27.500	2.500	37.500	32.500
Best Generation					
Average	55.050	124.325	55.550	131.800	135.725
Variance	455.382	2198.481	919.177	1663.805	2155.025
Standard Deviation	21.340	46.888	30.318	40.790	46.422
Run Time (sec)					
Average	4.470	4.296	10.143	11.641	7.699
Variance	0.000	0.000	0.113	1.531	0.863
Standard Deviation	0.014	0.004	0.337	1.237	0.929

Figure 93. Performance Statistics – Population Size 280

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0007114269	0.0000652606	0.0004305516	0.0000421857	0.0000360101
Variance	0.0000031068	0.0000000343	0.0000006309	0.0000000126	0.0000000122
Standard Deviation	0.0017626168	0.0001851510	0.0007942828	0.0001122424	0.0001103860
Best = Optimal	2	15	9	16	14
Percentage	5.000	37.500	22.500	40.000	35.000
Best Generation					
Average	50.975	119.825	60.775	148.775	134.075
Variance	529.256	1269.071	621.307	1707.102	1685.763
Standard Deviation	23.006	35.624	24.926	41.317	41.058
Run Time (sec)					
Average	5.091	4.921	11.421	12.957	8.390
Variance	0.000	0.000	0.220	1.669	1.368
Standard Deviation	0.013	0.005	0.469	1.292	1.169

Figure 94. Performance Statistics - Population Size 320

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0001484882	0.0000011267	0.0001565997	0.0000007502	0.0000008002
Variance	0.0000000668	0.0000000000	0.0000000664	0.0000000000	0.0000000000
Standard Deviation	0.0002583908	0.0000039916	0.0002577251	0.0000013161	0.0000013053
Best = Optimal	19	25	20	25	23
Percentage	47.500	62.500	50.000	62.500	57.500
Best Generation					
Average	64.125	135.100	64.900	148.850	136.250
Variance	1243.702	2143.067	893.631	1631.874	1611.936
Standard Deviation	35.266	46.293	29.894	40.396	40.149
Run Time (sec)					
Average	9.989	9.786	20.893	23.661	16.098
Variance	0.001	0.000	0.801	4.815	2.937
Standard Deviation	0.027	0.007	0.895	2.194	1.714

Figure 95. Performance Statistics - Population Size 640

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0000925564	0.0000005001	0.0001222803	0.0000011760	0.0000013523
Variance	0.0000000392	0.0000000000	0.0000000507	0.0000000000	0.0000000000
Standard Deviation	0.0001979426	0.0000009340	0.0002252211	0.0000029556	0.0000046322
Best = Optimal	28	26	25	25	28
Percentage	70.000	65.000	62.500	62.500	70.000
Best Generation					
Average	80.925	133.000	74.375	143.550	139.375
Variance	1566.225	2074.308	1397.369	1455.126	1567.420
Standard Deviation	39.576	45.545	37.381	38.146	39.591
Run Time (sec)					
Average	14.908	14.685	29.938	35.627	24.069
Variance	0.001	0.000	1.476	10.774	9.901
Standard Deviation	0.029	0.008	1.215	3.282	3.147

Figure 96. Performance Statistics - Population Size 960

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0000667514	0.0000013534	0.0000630746	0.0000007252	0.0000007253
Variance	0.0000000327	0.0000000000	0.0000000327	0.0000000000	0.0000000000
Standard Deviation	0.0001808587	0.0000057426	0.0001806995	0.0000013207	0.0000016340
Best = Optimal	32	30	34	26	25
Percentage	80.000	75.000	85.000	65.000	62.500
Best Generation					
Average	78.600	132.050	87.950	149.975	143.900
Variance	1112.759	1842.408	1572.459	1035.358	1437.272
Standard Deviation	33.358	42.923	39.654	32.177	37.911
Run Time (sec)	19.839	19.526	40.166	45.264	32.425
Variance	0.001	0.000	2.387	12.449	14.758
Standard Deviation	0.032	0.008	1.545	3.528	3.842

Figure 97. Performance Statistics - Population Size 1280

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0000370276	0.0000008516	0.0000334757	0.0000007252	0.0000006253
Variance	0.0000000178	0.0000000000	0.0000000175	0.0000000000	0.0000000000
Standard Deviation	0.0001333449	0.0000039488	0.0001322523	0.0000013207	0.0000016450
Best = Optimal	35	30	32	26	29
Percentage	87.500	75.000	80.000	65.000	72.500
Best Generation					
Average	70.750	133.225	95.650	135.175	141.500
Variance	965.782	2022.333	1991.721	1643.789	1539.333
Standard Deviation	31.077	44.970	44.629	40.544	39.234
Run Time (sec)					
Average	24.762	24.407	50.075	56.016	39.951
Variance	0.001	0.000	3.394	29.171	33.334
Standard Deviation	0.036	0.011	1.842	5.401	5.774

Figure 98. Performance Statistics – Population Size 1600

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0000036768	0.0000003001	0.0000223032	0.0000009759	0.0000004001
Variance	0.0000000005	0.0000000000	0.0000000096	0.0000000000	0.0000000000
Standard Deviation	0.0000230928	0.0000007235	0.0000981372	0.0000028907	0.0000007444
Best = Optimal	38	31	34	27	27
Percentage	95.000	77.500	85.000	67.500	67.500
Best Generation					
Average	77.600	142.025	83.275	147.675	146.550
Variance	1590.092	1467.615	2094.666	1461.097	1723.382
Standard Deviation	39.876	38.309	45.768	38.224	41.514
Run Time (sec)					
Average	29.687	29.263	59.905	68.432	45.822
Variance	0.001	0.000	10.219	59.578	23.450
Standard Deviation	0.034	0.010	3.197	7.719	4.842

Figure 99. Performance Statistics – Population Size 1920

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0000146074	0.0000005751	0.0000518770	0.0000008504	0.0000005501
Variance	0.0000000020	0.0000000000	0.0000000255	0.0000000000	0.0000000000
Standard Deviation	0.0000443803	0.0000010838	0.0001596033	0.0000017922	0.0000010853
Best = Optimal	36	26	34	26	27
Percentage	90.000	65.000	85.000	65.000	67.500
Best Generation					
Average	99.625	142.550	90.875	137.325	143.250
Variance	1291.676	1963.638	882.984	871.815	1701.936
Standard Deviation	35.940	44.313	29.715	29.527	41.255
Run Time (sec)					
Average	34.632	34.136	68.699	78.552	52.883
Variance	0.001	0.000	11.273	47.473	32.043
Standard Deviation	0.033	0.013	3.357	6.890	5.661

Figure 100. Performance Statistics - Population Size 2240

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0000148994	0.0000011517	0.0000074037	0.0000007002	0.0000007007
Variance	0.0000000088	0.0000000000	0.0000000010	0.0000000000	0.0000000000
Standard Deviation	0.0000939083	0.0000039875	0.0000322215	0.0000012030	0.0000025277
Best = Optimal	37	24	37	24	27
Percentage	92.500	60.000	92.500	60.000	67.500
Best Generation					
Average	77.425	134.725	92.775	138.950	143.225
Variance	671.122	1922.769	1698.948	2268.664	1840.846
Standard Deviation	25.906	43.849	41.218	47.630	42.905
Run Time (sec)					
Average	39.546	39.004	79.155	86.242	60.774
Variance	0.001	0.000	12.840	46.784	45.770
Standard Deviation	0.030	0.018	3.583	6.840	6.765

Figure 101. Performance Statistics - Population Size 2560

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0000148494	0.0000007257	0.0000036518	0.0000005502	0.0000010509
Variance	0.0000000088	0.0000000000	0.0000000005	0.0000000000	0.0000000000
Standard Deviation	0.0000939161	0.0000025256	0.0000230963	0.0000012189	0.0000028770
Best = Optimal	39	26	39	30	24
Percentage	97.500	65.000	97.500	75.000	60.000
Best Generation					
Average	92.875	120.825	90.725	141.875	143.150
Variance	1624.830	2284.097	1230.358	2028.933	1791.156
Standard Deviation	40.309	47.792	35.076	45.044	42.322
Run Time (sec)					
Average	44.489	43.868	88.465	97.880	68.463
Variance	0.002	0.000	20.858	73.952	46.018
Standard Deviation	0.039	0.017	4.567	8.600	6.784

Figure 102. Performance Statistics - Population Size 2880

Strategy	LS	LN	GS	GN	PN
Best Solution					
Average	0.0000000000	0.0000004251	0.0000148994	0.0000004751	0.0000005001
Variance	0.0000000000	0.0000000000	0.0000000088	0.0000000000	0.0000000000
Standard Deviation	0.0000000000	0.0000009309	0.0000939083	0.0000007509	0.0000010864
Best = Optimal	40	29	37	24	29
Percentage	100.000	72.500	92.500	60.000	72.500
Best Generation					
Average	93.375	131.500	87.900	137.875	133.925
Variance	1567.369	1747.385	1649.272	1445.599	2067.507
Standard Deviation	39.590	41.802	40.611	38.021	45.470
Run Time (sec)					
Average	49.401	48.735	97.023	106.952	76.524
Variance	0.001	0.000	19.807	99.388	161.663
Standard Deviation	0.030	0.018	4.451	9.969	12.715

Figure 103. Performance Statistics - Population Size 3200

Bibliography

1. Baker, James E. "Reducing bias and inefficiency in the selection algorithm." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. 14–21. Hillsdale NJ: Lawrence Erlbaum Associates, 1987.
2. Baker, James E. *Analysis of the effects of selection in genetic algorithms*. PhD dissertation, Vanderbilt University, Nashville TN, 1989.
3. Bennet, Kristin and others. "A Genetic Algorithm for Database Query Optimization." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 400–407. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.
4. Bethke, Albert D. *Genetic Algorithms as Function Optimizers*. PhD dissertation, The University of Michigan, Ann Arbor MI, 1980.
5. Bonomi, E. and J.L. Lutton. "The N-city traveling salesman problem: statistical mechanics and the Metropolis Algorithm," *SIAM Review*, *26*:551–569 (October 1984).
6. Bramlette, Mark A. and Rob Cusic. "A Comparative Evaluation of Search Methods Applied to Parametric Design of An Aircraft." *Proceedings of the Third International Conference on Genetic Algorithms*. 213–218. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.
7. Brassard, Giles and Paul Bratley. *Algorithmics: Theory and Practice* (First Edition). Englewood Cliffs NJ: Prentice Hall, 1988.
8. Bridges, C. L. and David E. Goldberg. "An analysis of a reordering operator on a GA-hard problem," *Biological Cybernetics*, *62*:397–405 (1990).
9. Brown, Donald E. and others. "A Parallel Genetic Heuristic for the Quadratic Assignment Problem." *Proceedings of the Third International Conference on Genetic Algorithms*. 406–415. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.
10. Caldwell, Craig and Victor S. Johnston. "Tracking a Criminal Suspect Through Face Space with a Genetic Algorithm." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 416–421. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.
11. Cleveland, G. A. and S. F. Smith. "Using genetic algorithms to schedule flow shop releases." *Proceedings of the Third International Conference on Genetic Algorithms*. 160–169. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.
12. Cochran, Will G. *Sampling Techniques* (Third Edition). New York: John Wiley & Sons, 1977.
13. Cohoon, J. P. and others. "A multi-population genetic algorithm for solving the k-partition problem on hyper-cubes." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 244–248. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.
14. Cohoon, J.P. and others. "Punctuated equilibria: a parallel genetic algorithm." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. 148–154. Hillsdale NJ: Lawrence Erlbaum Associates, Inc., 1987.
15. Conover, W.J. *Practical Nonparametric Statistics*. New York: John Wiley & Sons, 1971.
16. Das, Rajarshi and Darrell Whitley. "The only challenging problems are deceptive: global search by solving order-1 hyperplanes." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 166–173. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.

17. Davidor, Yuval. "A Genetic Algorithm Applied to Robot Trajectory Generation." *Handbook of Genetic Algorithms* edited by Lawrence Davis, 144–165, New York: Van Nostrand Reinhold, 1991.

18. Davis, Lawrence. "What is a Genetic Algorithm?." *Handbook of Genetic Algorithms* edited by Lawrence Davis, 1–22, New York: Van Nostrand Reinhold, 1991.

19. De Jong, Kenneth A. *Analysis of the behavior of a class of genetic adaptive systems*. PhD dissertation, The University of Michigan, Ann Arbor MI, 1975.

20. De Jong, Kenneth A. "Genetic algorithms: a 10 year perspective." *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*. 169–177. Hillsdale NJ: Lawrence Erlbaum Associates, 1988.

21. De Jong, Kenneth A. and William M. Spears. "Using Genetic Algorithms to Solve NP-Complete Problems." *Proceedings of the Third International Conference on Genetic Algorithms*. 124–132. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.

22. Dromey, Geoff. *Program Derivation: The Development of Programs from Specifications*. Sydney Australia: Addison-Wesley Publishing Company, Inc., 1989.

23. Eshelman, Larry J. and others. "Biases in a crossover landscape." *Proceedings of the Third International Conference on Genetic Algorithms*. 10–19. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.

24. Eshelman, Larry J. and J. David Schaffer. "Preventing premature convergence in genetic algorithms by preventing incest." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 115–122. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.

25. Forrest, Stephanie and Gottfried Mayer-Kress. "Genetic algorithms, nonlinear dynamical systems, and models of international security." *Handbook of Genetic Algorithms* edited by Lawrence Davis, 166–185, New York: Van Nostrand Reinhold, 1991.

26. Forrest, Stephanie and Melanie Mitchell. "The performance of genetic algorithms on Walsh polynomials: some anomalous results and their explanation." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 182–189. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.

27. Fox, Geoffrey C. and others. *Solving Problems on Concurrent Processors, 1*. Englewood Cliffs NJ: Prentice Hall, 1988.

28. Garey, Michael R. and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco CA: W. H. Freeman and Company, 1979.

29. Gehani, Narain. *C: An Advanced Introduction*. Rockville MD: Computer Science Press, 1985.

30. Goldberg, David E. *Optimal Initial Population Size for Binary-Coded Genetic Algorithms*. Technical Report, Tuscloosa AL: University of Alabama, 1985.

31. Goldberg, David E. "Simple Genetic Algorithms and the Minimal Deceptive Problem." *Genetic Algorithms and Simulated Annealing*, edited by Lawrence Davis. 74–88. London: Pitman Publishing, 1987.

32. Goldberg, David E. *Genetic algorithms and Walsh functions: Part I, a gentle introduction*. Technical Report, Tuscloosa AL: University of Alabama, 1988. TCGA Report No. 88006.

33. Goldberg, David E. "Genetic algorithms and Walsh functions: Part II, deception and its analysis," *Complex Systems*, 3:153–171 (1989).

34. Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading MA: Addison-Wesley Publishing Company, 1989.

35. Goldberg, David E. "Sizing Populations for Serial and Parallel Genetic Algorithms." *Proceedings of the Third International Conference on Genetic Algorithms*. 398–405. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.

36. Goldberg, David E. and others. "Messy Genetic Algorithms: Motivation, Analysis, and First Results," *Complex Systems*, *3*:493–530 (1989).

37. Goldberg, David E. and others. "Messy Genetic Algorithms Revisited," *Complex Systems*, *4*:415–444 (1990).

38. Goldberg, David E. and others. "Don't Worry, Be Messy." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 24–30. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.

39. Goldberg, David E. and Robert Lingle. "Alleles, loci, and the traveling salesman problem." *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*. 154–159. Hillsdale NJ: Lawrence Erlbaum Associates, 1988.

40. Grefenstette, John J. "Optimization of Control Parameters for Genetic Algorithms," *IEEE Transactions on Sytems, Man, and Cybernetics*, *1*:122–128 (1986).

41. Grefenstette, John J. *A User's Guide to Genesis*. Technical Report, Nashville TN: Vanderbilt University, 1986.

42. Grefenstette, John J. and James E. Baker. "How genetic algorithms work: a critical work at implicit parallelism." *Proceedings of the Third International Conference on Genetic Algorithms*. 20–27. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.

43. Grefenstette, John J. and others. "Genetic Algorithms for the Traveling Salesman Problem." *Proceedings of the Third International Conference on Genetic Algorithms*. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.

44. Grimaldi, Ralph P. *Discrete and Combinatorial Mathematics: An Applied Introduction* (Second Edition). Reading MA: Addison-Wesley Publishing Company, Inc., 1989.

45. Harp, Steven A. and Tariq Samad. "Genetic Synthesis of Neural Network Architecture." *Handbook of Genetic Algorithms* edited by Lawrence Davis, 202–221, New York: Van Nostrand Reinhold, 1991.

46. Hesser, J. and others. "Optimization of Steiner Trees Using Genetic Algorithms." *Proceedings of the Third International Conference on Genetic Algorithms*. 231–236. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.

47. Holland, John H. *Adaptation in Natural and Artificial Systems*. Ann Arbor MI: The University of Michigan Press, 1975.

48. Homaifar, Abdollah and others. "Analysis and design of a general GA deceptive problem." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 196–203. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.

49. Jog, Prasana and Dirk Van Gucht. "Parallelisation of probabilistic sequential search algorithms." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. 170–176. Hillsdale NJ: Lawrence Erlbaum Associates, Inc., 1987.

50. Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Englewood Cliffs NJ: Prentice-Hall, Inc., 1978.

51. Kleinbaum, David G. and Lawrence L. Kupper. *Applied Regression Analysis and Other Multivariable Methods*. North Scituate MA: Duxbury Press, 1978.

52. Knuth, Donald E. *The Art of Computer Programming* (Second Edition), *2*. Reading MA: Addison-Wesley Publishing Company, 1981.

53. Kruse, Robert L. *Data Structures and Program Design*. Englewood Cliffs NJ: Prentice-Hall, Inc., 1984.

54. Larsen, Richard J. and Morris L. Marx. *An Introduction to Mathematical Statistics and its Applications*. Englewood Cliffs NJ: Prentice-Hall, Inc., 1981.

55. Liepins, G. E. "Greedy Genetics." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. 90–99. Hillsdale NJ: Lawrence Erlbaum Associates, 1987.

56. Lucasius, C. B. and others. "A Genetic Algorithm for Conformational Analysis of DNA." *Handbook of Genetic Algorithms* edited by Lawrence Davis, 251–281, New York: Van Nostrand Reinhold, 1991.

57. Maybeck, Dr. Peter S. Personal interview. Air Force Institute of Technology, Wright-Patterson AFB OH, 10 Jan 1992.

58. Mendenhall, William and others. *Mathematical Statistics with Applications* (Fourth Edition). Boston: PWS-KENT Publishing Company, 1990.

59. Miller, Lawerence H. and Alexander E. Quilici. *Programming in C*. New York: John Wiley & Sons, 1986.

60. Muhlenbein, H. and others. "The parallel genetic algorithm as a function optimizer." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 271–278. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.

61. Nakano, Ryohei. "Conventional Genetic Algorithm for Job Shop Problems." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 474–479. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.

62. Office of Science and Technology Policy. *Grand Challenges: High Performance Computing and Communications*. Technical Report. Washington, D.C.: Committee on Physical, Mathematical, and Engineering Sciences, 1990.

63. Pettey, Chrisila B. and others. "A parallel genetic algorithm." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. 155–161. Hillsdale NJ: Lawrence Erlbaum Associates, Inc., 1987.

64. Pettey, Chrisila B. and Michael R. Leuze. "A theoretical investigation of a parallel genetic algorithm." *Proceedings of the Third International Conference on Genetic Algorithms*. 398–405. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.

65. Powell, David J. and others. "Interdigitation: A hybrid technique for engineering design optimization employing genetic algorithms, expert systems, and numerical optimization." *Handbook of Genetic Algorithms* edited by Lawrence Davis, 312–331, San Mateo CA: Van Nostrand Reinhold, 1991.

66. Pugh, Kenneth. *C Language for Programmers* (Second Edition). Wellesley MA: QED Information Sciences, Inc., 1990.

67. Ragsdale, Susann. *Parallel Programming*. New York: McGraw-Hill, Inc., 1991.

68. Riolo, Rick L. "Modeling Simple Human Category Learning with a Classifier System." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 324–333. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.

69. Rosenbrock, H. H. "An automatic method for finding the greatest or least value of a function," *Computer Journal*, *3*:175–184 (1960).

70. Sandgren, Eric. *The utility of nonlinear programming algorithms*. PhD dissertation, Purdue University, Lafayette IN, 1977.

71. Sawyer, George A. "Functional optimization using parallel genetic algorithms." *Compendium of Parallel Programs for the Intel iPSC Computers*, Volume I, Version 1.5, edited by Gary B. Lamont and others. Unpublished compendium. Air Force Institute of Technology, Wright-Patterson AFB OH, 1989.

72. Syswerda, Gilbert and Jeff Palmucci. "The Application of Genetic Algorithms to Resource Scheduling." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 502–508. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1991.

73. Tanese, Reiko. "Parallel genetic algorithms for a hypercube." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. 170–176. Hillsdale NJ: Lawrence Erlbaum Associates, Inc., 1987.

74. Tanese, Reiko. "Distributed Genetic Algorithms." *Proceedings of the Third International Conference on Genetic Algorithms*. 434–440. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1989.

75. Wilson, Kenneth G. "Grand Challenges of Computational Science," *Future Generations Computer Systems*, 5:171–191 (1989).

76. Yourdon, Edward. *Modern Structured Analysis*. Englewood Cliffs NJ: Yourdon Press, 1989.